VAlgoLang: Simplifying Manim in an Educational Setting

Group 19 — Supervisor: Dr Tony Field

Rini Banerjee (rb3018), Clara Lebbos (cl10918), Ilona Zhu (yiz18),
Mayank Sharma (ms6418), Arjun Banerjee (ab5518), Aditya Goel (ag3518)

Executive Summary

Visually presenting concepts has always been a powerful and successful technique for educators to explain them to their students. This is especially true in the domain of algorithms, where gaining an intuition for how a new algorithm works can be very challenging for an inexperienced student without some sort of concrete, visual walk-through.

With schools and universities going online all across the world over the last few months, visual education has taken on much greater importance with the absence of in-person interaction. However, no easily accessible tools currently exist that enable educators to flexibly and quickly create high quality visualisations of anything beyond the most basic algorithms. This project focuses on creating such a tool by enabling educators to visualise an algorithm merely by implementing the algorithm itself.

Manim is a powerful, shape-centric Python animation library that can be used to create educational animations of scientific concepts. Its major drawback is that it is "notoriously difficult to use" [1][2] and time-consuming because it is extremely low-level; this discourages almost all educators from being able to harness its educational potential. Yet, when wielded by an expert in Manim it can be extremely expressive and is well-received on Grant Sanderson's 3Blue1Brown YouTube channel which has amassed 3.37M subscribers.

VAlgoLang enables a user to write an implementation of an algorithm as they normally would and automatically generate a beautiful visualisation of this algorithm using Manim. The project introduces a new, simple language and an accompanying interpreter. The interpreter takes as input a VAlgoLang source file and produces a video animation that leverages Manim to visualise how the data structures and variables in the program change line by line. Additionally, the user can customise the final animation using code annotations and style properties.

Users have described the final software as incredibly easy to use as they do not have to think about low-level animation details. They have also found it to be highly flexible as the focus is on the algorithm, so any modifications can be propagated to the animation within seconds. Additionally, VAlgoLang provides extensive documentation on both the language and the various ways of interacting with it, including a web interface.

There are immediate applications for VAlgoLang both at Imperial College London and beyond. At Imperial, the tool has applications in tutorial sessions for first year students, as well as in lectures. More broadly, independent learners can use VAlgoLang for visualising solutions to algorithms problems, such as those on software engineering interview practice platforms like LeetCode. The project has been open sourced and has gained some traction among Manim enthusiasts on Reddit. Going forward, the group is excited to see how far the project can go.

1 Introduction

Representing computer science concepts visually makes teaching and understanding them easier [3], and a variety of solutions exist to help educators produce animations that achieve this goal. Some may like to use PowerPoint, which is simple enough to operate but can be difficult to modify as the complexity of the animation increases. Others may prefer to code up their own animations using third-party animation libraries, but this can be quite challenging. A mathematician, Grant Sanderson, created a Python animation library called Manim [4] which he uses to generate videos for his YouTube channel 3Blue1Brown [5]. His channel includes several topics ranging from computer science topics to complex mathematical theorems, all of which are supported by beautiful visuals.

Manim is primarily used by enthusiasts and YouTubers to create their own animations. While Manim can be easily accessed as a Python library, creating a short video can require hundreds of lines of low level, animation-focused code. Additionally, two versions of the library exist: Manim and Manim Community; the latter was created by individuals and is community-maintained, and was developed due to a lack of documentation on the original library. The lack of documentation, amount of dependencies, as well as the large amount of Python code required make it challenging for individuals to use Manim. For instance, his most recent video on Hamming Codes took 6270 lines of Python to develop. In fact, Grant Sanderson "originally put it together as a somewhat scrappy personal project, more for my [his] own use cases than anything professionally maintained or explicitly outward-facing" [6].

The aim of this project is to simplify the process of creating educational animations using Manim for Computer Science topics, specifically data structures and algorithms. This entails developing an accompanying interpreter for a new programming language (called VAlgoLang) that generates Python code using the Manim library. The user has two options: use the interpreter locally or via the web interface. For the first option, the user can install the interpreter, input their algorithm written in VAlgoLang, and this will generate the visualisation of the data structure(s) line-by-line as the code executes, as well as, optionally, the corresponding Python code. Alternatively, the user can use the web interface to import their project or write their algorithm and similarly generate the corresponding video and Python code. An important goal of the project was to give the user a lot of control over the animation: from colour to animation, speed and placement of visual elements in the video. The aim was to allow the user to modify the video without having to dive into the lengthy Python code.

The next sections contain explanations on the design and implementation choices that were made, followed by the technical challenges faced. Then, there is an evaluation of the tool that was built and a discussion regarding how it solves the original problem, before an exploration of ethical issues.

1.1 Manim Overview

Manim is an open source Python library used to create animations programmatically. Its main building blocks are called `MObjects` (Manim Objects) which represent most basic shapes, such as: Rectangles, Circles, Arrows, Arcs and Text boxes. The main class of the Python file being used to generate a Manim video should inherit from Manim's Scene class; each video can only represent one Scene. Within the main class, `MObjects` can be added to the Scene and positioned using the coordinate system. The Scene has an x-axis ranging from -7 to 7, and a y-axis from -4 to 4, with the point $(0, 0)$ being the centre of the Scene. Another way to position `MObjects` is by placing them relative to others (above, under, to the right or to the left). It is important to note that Manim does not handle scaling: it is valid to place a `MObject` at coordinates $(30, 60)$, but it will not appear in the video in this case.

2 Design & Implementation

2.1 Overview

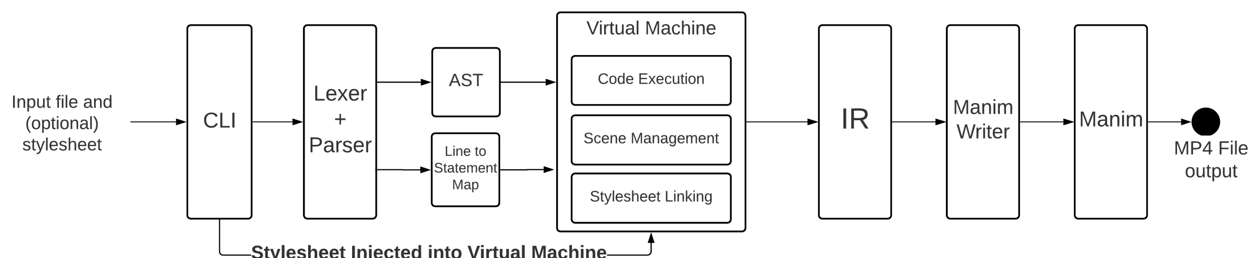

Figure 1: Implementation Overview

Figure 1 shows an overview of the project implementation. The user uses the Command Line Interface (CLI) to interact with the interpreter, passing in relevant arguments such as the location of input code file and animation quality. The input code is then traversed to build an internal abstract syntax tree (AST). Any syntax error is thrown during parsing and any semantic error is thrown during AST generation. If the AST is generated successfully, the code is executed in a Virtual Machine, which would produce the Intermediate Representation (Linear Representation) by keeping track of variables and frames. Finally, the Linear Representation is converted to a Python file written using Manim. This Python file is then executed, producing the output video file in the location specified by the user.

2.2 Software Engineering Practices

In order to guarantee high quality code and good development, the team set out a few rules to follow. Extreme programming (XP) was practised throughout the project, with elements of Kanban introduced where necessary. In practice, the team carried out weekly iterations with a planning session earlier on in the week, as well as daily stand-ups. To maintain similar knowledge across the stack amongst members of the team, during these sessions, all team members were updated on what is being worked on and shared documentation. Also, during the planning sessions, to gauge priority and length of work per feature/task, the team practised technological spikes.

To maintain high code quality and implement continuous integration, test-driven development was carried out, documentation was kept up-to-date and thorough code reviews were conducted. This was to ensure code could be released often and with confidence. Indeed, the team decided documentation was essential to this project, and needed to be written as soon as the project began, instead of during the last iteration. The documentation can be found here, and it is regularly updated with every Pull Request made from a contributor.

As an open-source project, the source code had to be well structured and well documented for newcomers. Therefore, the team made sure the code was modular in design, enabling contributors to add new data structures to the language by adding code in new files. KDoc [7] was also used to generate detailed documentation of all the language level features (e.g. every class constructor and method) from the code comments. This makes the code much easier to navigate and understand. The source code documentation can be found here.

2.3 Design Choice: Developing A Novel Interpreter

The choice of whether to write a novel interpreter or to develop a library for an existing language like Python was an important early decision. The desire was to allow users to write an implementation of an algorithm as normal and, with no real extra effort, build a great animation of the execution of their program using the solution.

The two major components our tool required to make this happen were as follows:

1. It must the program in the same way a normal runtime environment would, which entails executing lines of code and updating the call stack as appropriate.
2. It has to produce the animation itself based on the execution, which is done by generating and executing Python code that leverages some basic pre-built Python libraries as well as the Manim animation library.

It was quickly realised that executing the user's source code would not be optimally achieved through a library and so utilising an interpreter with a virtual machine to hold state about the execution of the user's program was the best option. It was also clear that the interpreter would require some novel compiler intermediate representations to be able to generate a visualisation of the user's code being executed. Existing interpreters focus on executing the user's source code efficiently; therefore, their internal data structures and intermediate representations are not suitable for code visualisation. It was thus concluded that the best approach would be to develop an interpreter from scratch.

Finally, developing a library for an existing language ran the danger of making the user reason about the animation at a lower level of abstraction than we wanted. By developing an interpreter, it was possible to ensure that all the user had to think about was the algorithm itself, letting the interpreter do the heavy lifting of producing the animation.

2.4 Design Choice: Creating A New Programming Language

While it was a relatively simple decision to write the interpreter from scratch, deciding whether to create a new programming language or implement a subset of an existing one was less straightforward.

One alternative to making a new language was to take an existing language like Python and develop a novel interpreter for a subset of it. Although this approach would mean a wider group of people could immediately use the tool, it was thought that doing so would slow down development at the start of the project. In addition, losing control over the syntax and semantics of the language would make development of the interpreter complex. Having control of the semantics is especially useful, as it allows decisions to be made that are most convenient for the purposes of producing a useful animation. A common approach to teaching algorithms is to demonstrate them in a language-agnostic manner - the new language has a concise, simple syntax to put the focus on the algorithms.

Adding support for user-defined syntax bindings is something that has been considered as possible future work for users that would like to write using a specific language's syntax.

2.5 Language Choices

As a group, there was a priority to effectively and efficiently use development time. Therefore, languages and frameworks were chosen based on the familiarity to the group to enable faster development.

To implement the interpreter, Kotlin [8] was chosen for the following reasons: firstly, Kotlin is a language which runs on the Java Virtual Machine (JVM), meaning the interpreter can be packaged and distributed easily. This is because the JVM allows Kotlin to be platform independent and hence can be used on any machine with a JVM installed. Secondly, Kotlin's conciseness (`when` blocks, data classes, smart casting) over Java leads to faster and neater development. Finally, Kotlin is inter-operable with Java, so there can be utilisation of Java libraries and the generated parser classes produced by the parser generator (ANTLR4 [9]).

ANTLR4 was chosen as the parser generator to implement VAlgoLang's grammar and generate the lexer and parser classes. An advantage of ANTLR4 is that from the same grammar, one is able to generate lexer and parser classes for multiple target languages. This feature was essential for the interpreter in Kotlin (with Java as the target language) as well as the web editor (with JavaScript as the target language). The latter

enables the feature of live syntax errors using the parse tree (see Section 2.6.1) in the web editor.

The web editor consists of a React [10] application frontend and an Express.js [11] backend, both of which are written in TypeScript [12]. The choice of TypeScript over JavaScript is due to its static type-checking which is easier to debug than JavaScript, as well as inter-operability with JavaScript for using the ANTLR4 generated parser.

The React framework was chosen due to its ease of use, and component-based organisation both of which enable fast and concise development. For the editor window itself, Microsoft's open source Monaco Editor is used [13]. This is because it is the editor powering the well-known code editor Visual Studio Code and there is also a packaged React component available to easily integrate it into the web editor frontend application.

The backend application, which runs the animation generation tasks using the interpreter through a REST API, is written in Express.js as it is lightweight and simple - perfect for the simple application. An alternative to this would be Spring; however, Spring offers multithreading as well as finer control to server-side processes leading to greater complexity - both of these factors are unnecessary for the simple backend application.

2.6 Technical breakdown

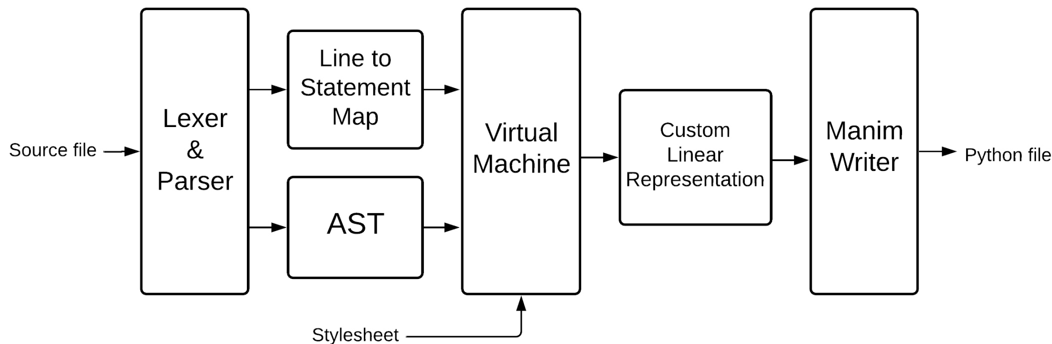

Figure 2: Interpreter Phases

2.6.1 Interpreter Frontend

One of the aims of this project was to create a familiar-feeling and easy-to-use DSL. Therefore, a simple grammar in ANTLR4 was designed with inspiration from Kotlin. The language tour is available here.

The first stages in the interpreter are lexical and syntax analysis (parsing the input file) to produce a parse tree. From the grammar, ANTLR4 generates the parser class which essentially does this task. Syntax errors are thrown using a custom syntax error handler, allowing the interpreter to produce meaningful error messages with additional help, such as underlining where in the line the error has occurred.

After the parse tree has been produced, the compiler simultaneously performs semantic analysis and constructs an abstract syntax tree (AST) in one pass. To traverse the parse tree, the visitor pattern is followed, extending the generated parser visitor abstract class and ultimately producing the AST node representing the entire program. The visitor pattern has the advantage over the listener pattern of enabling control over how the tree is traversed and also allows for return values. This means the interpreter can more easily perform context-specific semantic checks, such as checking the argument types in a function call just after visiting the argument expressions as well as easily constructing nested nodes. The alternative to this is the

listener pattern; this pattern automatically visits each node in the tree and calls two functions per node, an "enter" and "exit". This has the downsides of being lengthy code and not maintaining the ability to return values, so constructing nested nodes for an AST is more complex.

To implement semantic checks, a symbol table was needed in order to store variable names, function names and their respective types. The symbol table was designed as a tree of scopes, where the root node was the global scope. This meant whenever a new code block was entered (function call, branching statement, loop) a new scope was entered, which appended a child scope onto the current scope and updated the current scope to the newly added child scope. To ensure a variable declaration is valid, the symbol table visitor checks the existence of the variable's name in the current scope. If it is not present, the parent scopes are recursively checked for this variable name until the global scope is reached. If the symbol table does not contain the variable name, the declaration is added to the current scope. Similarly, if a variable is referenced and is not present in the symbol table a semantic error is thrown detailing the undeclared variable. Once the end of the block has been reached, the current scope is set to be its parent scope. If semantic errors are detected, the interpreter halts at this stage and produces meaningful error messages to the user.

While the AST is being constructed, an auxiliary data structure called the Line to Statement Map (LSM) is produced. This is a map of line numbers in the input file to AST statement nodes. The LSM is used to sequentially execute lines of code later on in the Virtual Machine (see Section 2.6.2). This structure is created here because the parse tree contains contextual information such as the line number.

The type system for VAlgoLang is largely based on two categories:

1. **Primitive Types**: these represent basic types that do not require complex animation and are more for algorithm execution. There is currently support for strings, characters, boolean values and number values (any real number with double precision).
2. **Data Structure Types**: these represent more complex types that are animated. These types are baked into the compiler due to complexities in handling their values in algorithm execution in the Virtual Machine (see Section 2.6.2) and animating them. The superclass for data structure types can be extended to introduce more data structures into the language. Details on this can be seen in the List case study.

2.6.2 Code Execution

The Virtual Machine models a stack machine and is used to execute the user's program. The LSM constructed earlier maps unique line numbers to Statement Nodes of the Abstract Syntax Tree. The Virtual Machine first constructs a `Frame` for the entry point of the program - as the name suggests, this models a section of the call stack. Running the `Frame` from a start line number to an end line number entails executing each Statement in the LSM until the end line number or a return statement is reached.

For function calls, a new `Frame` is created and run, with start and end line numbers extracted from the Statement representing the body of the function call. The return value of running the frame is used to propagate values up the call stack and is the means by which recursion and therefore loops are modelled.

Scoping was handled by keeping the current value of a variable during execution local to the `Frame` and not to the overall Virtual Machine. When the scope changes (e.g. during a function call or when entering the body of a loop), a new child frame is created. The child frame created may be pre-loaded with the arguments from a parent frame. For instance, when creating a child frame for a function call, values for function arguments are passed through, and in a loop the parent scope's values are passed through.

Data structures are "passed by reference". Data structures created locally in each scope are specific to the `Frame` and are recorded in a set. When the program execution exits the child scope, the local data structures that are not being returned will be "destroyed" and no longer show up in the animation. Furthermore, the most recently used "live variables" at any given state are used to update the `Variable Block` (see

Section 2.6.6) visible on the top left of the animation if enabled by the user. Returned data structures are modelled as being "heap" allocated and so have lifetimes that persist beyond the ending of the frame they were constructed in. Basic copy constructors are defined for this in the Virtual Machine, like those in C++.

If something goes wrong at runtime, such as indexing an array out of bounds, the compiler catches this and produces "runtime errors", which are propagated up the active frames to the encompassing Virtual Machine, stopping the program early and informing the user.

```
let arr = Array<number>(3);
let value = arr[3]; // Out of bounds
```

Figure 3: Indexing an array out of bounds

Executing the above code will index the fourth element of an array of size 3. The following runtime error is produced:

```
Compiling...
Error detected during program execution. Animation could not be generated
Exit code: 300
Your program failed at line 2: Array index out of bounds

Process finished with exit code 44
```

Figure 4: Indexing an array out of bounds runtime error

The benefit of having all of this contextual information in the Virtual Machine (or one of its `Frames`) is that with each statement execution, the Linear Representation can be built up. The user-defined stylesheet is also used for this. The information retained by the Virtual Machine is also used to automatically size and place elements on the Scene. There are sections devoted to both of these later. Details about all the error types can be found in the documentation here.

2.6.3 Controlling the Final Animation

User feedback has made it clear that an important aspect of this software would be to give the user control over how the final animation looks. In order to achieve this goal, a stylesheet has been introduced: a file in which the user can customise various stylistic attributes of their Manim video in a simple and intuitive way.

The stylesheet is a JSON file which is parsed after syntax and semantic analysis (see Section 2.7.1). Once the custom attributes have been retrieved from the stylesheet, they are passed into the virtual machine and incorporated into the linear representation. The stylesheet is entirely optional, so if the user does not provide a stylesheet, a set of default style and animation properties are applied to the generated animation instead.

The customisable attributes in the stylesheet can be applied to specific data structures in the code by variable name, or to all variables that are of a particular data structure type (e.g., all stacks in the program). These style attributes include properties such as the colours of the text and the border of data structures, both "at rest" and while being animated (e.g., the top element of a stack changing colour during the "pop" operation). The user also has control over the amount of time a data structure takes to be created or animated, as well as the Manim animation effects used when a data structure is created or animated. The stylesheet also gives the user control over the shape and location of the code block, variable block and each of their data structures in the final animation, simply by specifying the coordinates of the bottom left-hand corner and the desired width and height for each area of the screen. For full details of what types of style properties are available, please refer to this page of the documentation.

Styling is applied using the following hierarchy: variable style takes precedence over the style for its data structure type, and this in turn takes precedence over the default style properties. For an example of this,

please refer to this page of the documentation.

While the stylesheet provides a way for users to statically specify the look of their animation, the need to control specific parts of the code became apparent very quickly. To address this, users are given the option to add annotations to further control the animation. For example, users can speed up or slow down parts of the code, pause the video at certain steps, dictate when to step into or over a block of code (much like a debugger), and add subtitles to explain the process of certain steps as they are visualised. All of this means users are able to control very specific parts of their algorithm to show up a certain way in the final animation.

Note: Annotations do not appear in the final animation for code tracking (see Section 2.6.6).

2.6.4 Linear Representation

As mentioned previously in the Code Execution section (see section 2.6.2), the virtual machine is used to build up the linear representation. Each instruction implements an abstract method that returns the Python code corresponding to that linear representation instruction.

Broadly speaking, there are two types of instructions in the linear representation. The first type of instruction involves constructing something on the Manim Scene for the first time. One subset that falls under this category are the instructions that are needed to create new objects on the Scene, which need to be assigned to a particular location in the final animation by the boundary assignment algorithm (see Section 2.8.1). These include not only the animation objects needed to construct data structures for the first time (e.g., the root of a tree or the 'base' of a stack) but also other objects that need to be allocated space by the algorithm, such as the code and variable blocks. Another type of "object-constructing" instruction is for those objects that are new on the Scene but are associated with an existing data structure and, therefore, do not need to be allocated any space themselves. An example of such an instruction would be anything that can be dynamically added to a data structure, such as the child nodes that are added to the root of a tree after it has been initialised, or the objects that are pushed onto a stack.

The second type of linear representation instruction involves animating operations on objects that already exist on the Scene. This includes instructions that restyle pre-existing objects, push and pop operations on a stack, etc. Many instructions are open-ended and extensible, and so anyone interested in extending VAlgoLang with more data structures could use these if they wished.

In addition, styling is incorporated into the linear representation by passing through the corresponding style attributes. Each object that is assigned boundaries in the space allocation algorithm contains the static styling attributes for that component of the animation - for instance, the most common style attributes color and text color. On the other hand, styling attributes specific to animating operations, such as the animation style for indicating array access, are extracted from the stylesheet if the user has defined them and passed to the restyle instructions.

2.6.5 Python Code Generation

The final stage of the interpreter is turning the Linear Representation into Python code, which is then executed to generate the output animation. To make the generated Python file more structured and readable, a few Python libraries have been created and are used to visualise and display different components of the animation - the data structures, the input code, subtitles and the most recently used "live variables". In addition, there are also utility functions that can be reused to generate repeated animations, such as moving the code tracking pointer to different lines of the code.

The output code is generated by converting each Linear Representation Instruction to Python code. To generate the entire Python file to be executed, the file is first prefilled with the dependencies and main function declaration used to generate the animation. Then, each instruction's Python code is appended with indentation. Finally, the utility functions and required prebuilt Python libraries are copied into the output file for it to be fully compilable. Moreover, to help the user better understand the connection between

generated Python code and the original input code, comments are generated along with the output code, describing the role of each block of generated code in the animation.

The main reason for the abstraction of components of the animation into prebuilt Python libraries is to make the output file more organized and readable. In particular, as the Manim library requires all the animation code to be in one class, the layer of abstraction reduces code duplication, and thus making it easier for the user to navigate through the generated code. For instance, if the user declares several instances of the same data structure, this simply results in several instances of the same prebuilt Python class being created, which avoids duplication of the Manim code.

Furthermore, an abstract class representing data structures was created, which utilises inheritance to further reduce code duplication. As data structures share some common attributes and methods needed to generate animations, these fields and functions are defined in the superclass and functions specific to each data structure are implemented in the subclasses.

In addition, a few building blocks such as a rectangle shape with text inside are added, which are used to visualise some data structures. These building blocks have been created with the intention to help the user visualise any user-defined data structure if they find them useful. Together with the prebuilt libraries for other components and the utility functions, this layer of abstraction leaves room for the user to extend the output of the compiler. For instance, one can easily change any predefined default visualisation of different data structures or add in their own if they want to.

2.6.6 Code and Variable Blocks

One of the main aims of this project is to help the user visualise how the data structures and variables in their program change line by line. In order to achieve this, it became evident that the video should not only contain the user's code, but also some way of tracking which line of the program is being executed at any point. The interpreter already has the Line to Statement Map, which is built at the same time as the AST (see Section 2.6.1) and is passed into the Virtual Machine. There, the statements which need to be shown in the final video (see Section 2.6.3) are passed onto the code block being rendered in the animation.

An important consideration is the legibility of the code in the video. If the user's code is longer than about 15 lines, rendering the entire program would make it difficult to read, even when the full height of the Manim Scene is utilised. To tackle this issue, scrolling within the code block is used; this involves showing a fixed number of lines of code on screen (depending on the dimensions the user has defined for the code block), and "scrolling" up and down when the program is executing a line of code that is offscreen.

Users have also suggested a visualisation of the execution of their program in real time would be useful, showing the state of the variables at each point in the program. It is important that the user is not overloaded with information and the variables displayed are relevant to the line of code being executed. This is achieved by tracking the most recently updated variables using a FIFO queue of variable indices. The variable block is updated whenever a variable is declared or reassigned.

2.7 Online editor

As an addition to the main compiler with a CLI, an online editor has been developed with full functionality for compiling, viewing and downloading compiled videos. The dependencies for Manim can grow to as large as 6GB; therefore, the aim is to make the software as accessible as possible for end users. Developing fully containerized Docker images (see Section 2.7.3) is one step in that direction, which means that users could pull an image from Docker Hub [14] to their computer and use it without having to install individual dependencies. This makes it easier for users to start using VAlgoLang, but the size of the image is still large and so users may not adopt it.

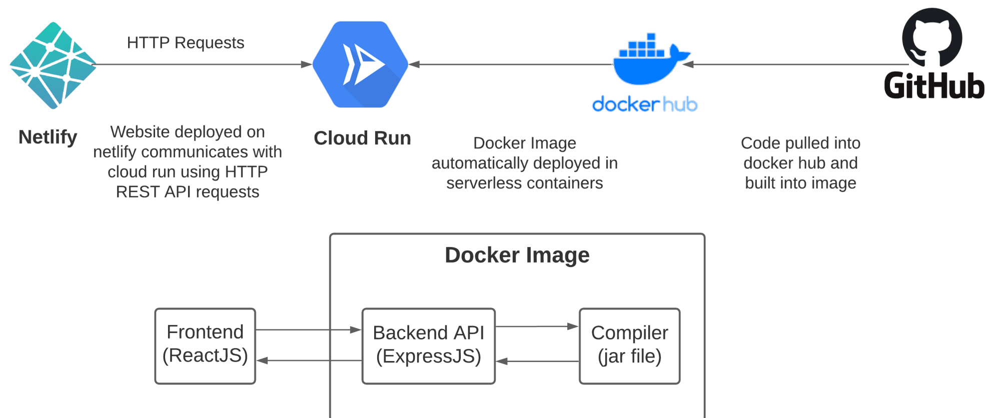

Figure 5: Online Editor Architecture

As a solution to this, an online editor has been created, where users can import projects, edit their code on the web editor, and compile it all from within a browser. The request to compile is sent to a server which has all the dependencies installed. The server receives the code, compiles it, and sends back the mp4 for the user to preview and download.

Another reason for developing the online editor is to allow the user to directly visualise some stylesheet properties - particularly the positioning of different components. The sizes and positioning supported by the stylesheet are based on a 2D coordinate system; however this may not be the most intuitive for the user. The online editor gives the user the option to visualise the placement of code and data structures and directly change the size and position of each, before generating the output mp4. The corresponding coordinates are updated in within the stylesheet in the online editor in real-time. This means that the user can then continue editing or compiling the code and the stylesheet, so that they do not have to manually calculate the sizes and coordinates.

The architecture is explained in the following sections.

2.7.1 Frontend

The frontend is a basic React application. The components available to the user are:

1. Interactive web editor, with syntax highlighting and live syntax errors
2. Option to validate code which checks and reports and semantic and runtime errors
3. Variable placement: Allow users an interactive way to modify exactly how big data structures are and where they should be positioned (see Figure 7)
4. Import project and download projects as zip for later use
5. Compile the project and download MP4 and Python files

The frontend is deployed on a free hosting site called Netlify [15]. This was chosen as they provide Asset Optimizations, CDN deployment and Continuous Integration along with a memorable domain name: `valgolang.netlify.app`

Figure 6: Web Interface

2.7.2 Backend

The backend is a simple Express server (written using TypeScript). Whilst used mainly by the frontend UI, the backend API is a standalone project, allowing users to send API calls themselves if they wish. It consists of 3 main endpoints:

1. /compile : Main endpoint which will save the le provided, pass it to the compiler using the CLI commands, and return a uid for the process running in the background, responsible for compiling and returning the right information.
2. /status : Endpoint used to track the status of each \job" based on the uid returned from the /compile endpoint. This endpoint returns the MP4 le once the process completes.
3. /boundaries : Endpoint used to return the auto-calculated boundaries for each data structure shown on the Scene. This is given as the \default" for the user to see when they use the placement manager to decide where they want to place everything.

2.7.3 Docker

Docker [16] is extensively used for deployment of the Web UI. This is because one of the goals of the project is to give the user as many ways as possible to interact with the software with the means they nd easiest. Pulling a Docker image locally on their computer and running the server may be more convenient for users (especially for o ine access).

Multiple Docker Images are built and published on Docker Hub here, and these Docker Images are exactly what are used for the main deployment of the software. A service from Google called Cloud Run [17] is used, which deploys Docker Images using a serverless architecture. The serverless architecture was chosen as it automatically scales up and down based on the number of users, which is critical as the compile time for algorithms can take a few minutes, and so many instances of the compiler is preferable to one.

Cloud Run pulls the code from GitHub, builds the latest Docker Image, and automatically deploys on every commit to master, meaning that as features are added, the web users of VAlgoLang always get access to the latest version.

2.8 Technical challenges

2.8.1 Placement of Data Structures

One challenge to overcome was placing the data structures the user defines in their code. For this, a simple algorithm was designed to place different types of data structures in the right place, which also took into account how large the data structure will grow. The virtual machine has access to all this information, and so each data structure's maximum size was tracked. Data structures with a higher maximum size were allocated more space than others.

The algorithm in a high level works as follows:

1. Each data structure is assigned one of 3 categories: Tall, Wide and Square, depending on the shape of the structure. For example, stacks are assigned Tall shapes, 1D Arrays are given Wide shapes, and 2D Arrays are given Square shapes.
2. Each Shape also defines the minimum dimensions of the shape on the Scene. to prevent data structures from being made too small.
3. As the code executes in the virtual machine, every construction of a new data structure is tracked with an ID and every operation on it updates the max size of the data structure. For example, with the stack, every push method increments the max size for the shape, but the pop method does not decrement this.
4. The information collected is then sent into the algorithm, which calculates the 4 coordinates representing each data structure's position on the Scene. Data structures are first placed using their predefined minimum dimensions. Unless there are too many data structures to fit on the scene (which throws a runtime error), the process of making shapes bigger commences as follows:
5. Each shape (Tall, Wide or Square) holds information about how the shape can expand. Tall shapes expand by increasing their height, Wide shapes expand by increasing their width, and Squares expand by increasing both their width and height. Based on the max sizes, shapes attempt to expand themselves, with the largest data structure attempting first.
6. Finally, once all the shapes have expanded as much as they can, all shapes are centralised both vertically and horizontally as it was assumed to be most desirable.

The result of this are presented below:

Figure 7: Auto placement with an array and 2 stacks

Here, the array is of size 10, and bothstack1 and stack2 have a max size of 3. This means the array is expanded rst, giving it the most space possible length-ways (excluding the code and variables).

Of course, this is always the \best guess", and so there are going to be cases where the user wants completely di erent positioning to what is auto-calculated. To account for this, a feature was added to the Web editor which allows the user to drag around and resize all the shapes however much they want and then compile the video, giving them full exibility over the nal placement of all the components in the video.

2.8.2 Scaling

As Manim does not handle automatic scaling (see Section 1.1), ensuring data structures did not grow out of their boundaries while maximising the use of their allocated space was a challenge. To ensure data structures automatically scale, an abstract base class was created, which has several scaling methods, both to check for over owing and to scale elements. Each data structure class (Stack, Tree, Array) inherits from this class. As Arrays are xed in size, the scaling is done only once when the array is created. The main challenge lies in scaling data structures when they grow out of the Scene, particularly for Stacks and Trees.

For Stacks, since they can only grow vertically, scaling mainly relies on the top boundary and the height of each stack and the new item to be pushed onto the stack. When pushing onto the stack, before actually animating the push operation, the rst thing checked is whether pushing the new item onto the stack will cross the top boundary allocated to it. If this is the case, the existing stack is scaled down to a level such that the stack will stay within the boundary with the new item pushed on top. Once the existing stack and the new item created are scaled, the creation of the item and the push operation are animated.

When popping o the stack, the removal of the topmost item of the stack is animated rst (with di erent animation depending on if the pop instruction is wrapped in a push instruction onto another stack). Then, if needed, the stack is scaled back up to fully utilise the allocated space.

For Trees, the challenge was handling horizontal growth in addition to vertical growth. Trees grow downwards and outwards, hence scaling mainly relies on the side boundaries and bottom boundary. To tackle this, two main cases have been considered.

Firstly, appending a new node onto the tree could result in the following two issues.

1. Over owing : this means that the child node is out of bounds. To detect this, prior to adding the child node to the tree, the rst check that is done is to make sure adding a child node does not cross any boundaries de ned for the Tree. If it does, the tree is scaled by using the minimum ratio between tree width, boundary width and tree height, boundary height. Once scaling is complete, the tree is translated to be positioned in the centre of its boundaries.
2. Overlapping subtrees : this occurs when the left subtree and right subtree overlap. This is detected recursively by traversing the tree depth- rst from the root and checking for each node that the rightmost edge in the left subtree does not cross the leftmost edge in the right subtree. If an overlap is detected, each subtree is translated away from each other until they do not cross over. Doing this increases the width of the tree, so an over owing check is run afterwards.

Secondly, when removing nodes from the tree, an under ow can occur. This is achieved by the tree not su ciently lling the space it was allocated. When a node is removed, a similar check to over owing is done by calculating the ratios between tree and boundary dimensions. In the case that there is under ow, one of these ratios will be greater than 1. The tree is then scaled according to this ratio and translated to the centre of its boundaries.

2.8.3 Keeping Track of Data Structures

When it comes to updating or reassigning variables that are data structures, the variable map that maps each variable identi er to its executed value is used. Whenever a data structure variable is reassigned, the

corresponding data structure is faded out and its value is cleared in the Virtual Machine's variable map. Following this, the program execution continues and the new data structure is constructed and rendered.

For data structures created locally in each scope (such as in functions), instructions are added that fades out all of these locally created data structures when exiting the scope. In particular, for any data structure created locally in a function and returned, a copy of the data structure is constructed and animated for the variable assigned to the function call. However, for data structures returned from recursive functions, if the same logic was followed, a new data structure would be copied over each recursive call, taking up excessive amounts of space in the animation. Therefore, for recursive function calls, the same space is reused on the Scene as for the child call - this particular information is available to the interpreter from the frame.

3 Evaluation

3.1 Testing

In order to ensure the quality and functionality of the software, the software has been tested thoroughly using JUnit, a unit testing framework designed for Java and Kotlin. Testing is mainly done by taking advantage of parameterized tests, testing for syntax errors (by checking for the 100 exit code), semantic errors (200 exit code), and runtime errors (300 exit code). Along with the right exit code, tests are also done to make sure the correct error message is thrown.

Along with invalid tests, valid tests are also conducted to test for all edge cases of the language. This meant that any new changes were tested on a large suite of tests, making sure any additions did not break any previous features. Moreover, there are also unit tests that cover the di erent stages of the interpreter in isolation such as the linear representation along with overall integration tests from VAlgoLang code to Python.

In total there are 252 tests, with 90% code coverage, determined using the IntelliJ code coverage runner. There is also frontend testing using mock responses and Jest [18] to make sure the frontend works as expected.

3.2 Ease Of Use

One of the main goals for this project is to simplify the process of using Manim and to ensure that VAlgoLang is considerably easier and more user friendly to use than Manim. The extent to which this goal has been met is veri ed through exploring the user journey and collecting corresponding metrics.

3.2.1 Old vs New User Journey

A considerable amount of e ort is required to generate even a simple animation using only Manim. This is made clear in Figure 8, which depicts the user journey for an educator who would like to generate a Manim video without using our tool. The large amount of time and e ort needed to be able to generate an animation can be discouraging to those who are unfamiliar with Manim.

Figure 8: Old User Journey

- Downloading Manim and its dependencies takes up a lot of space (up to 6GB) as well as time. It can be di cult to install depending on the operating system being used [2].
- The documentation for Manim is extremely scarce [1], and so users of Manim need to rely on searching through GitHub issues, reading or watching tutorials online and asking for help from the online community. All of these are extremely time-consuming and there is no guarantee that a user's particular issue will be resolved quickly.
- We found that the average amount of time taken to run Manim on a few test algorithms was quite high (about 87 seconds, as found in Table 4), and so potential users may be discouraged if they have to run Manim every time they make a small change and want to check the animation is still performing as expected.

Our solution aims to make this process much shorter and more approachable to new users. The new user journey for educators using VAlgoLang is shown in Figure 9.

Figure 9: New User Journey

Overall, we have tried to improve the the user experience at each step of their journey:

- If the user chooses to use the Web UI, there is no need to install Manim and all of its associated dependencies, which saves time and space. Alternatively, the user always has the option of working locally if they so wish. The compiler itself is very lightweight (less than 10MB), and so if a user already has Manim and its dependencies installed, it will not take them much more e ort to install VAlgoLang.
- Once the user has set up the environment they wish to work in, all they have to do is read the quick start guide to understand how the language works. The language is simple enough that anyone with basic programming experience should be able to pick it up fairly quickly.
- The focus is on the algorithm itself rather than how it should be visualised, and so the user simply needs to write the code for this algorithm in order to produce a Manim animation of it.
- Customising the nal animation is much simpler using the stylesheet, annotations and command-line options.

3.2.2 Ease Of Use Metrics

This section presents two proxies to measure the ease of use. Firstly, an investigation on the number of lines of code that a user has to write to produce equivalent animations in VAlgoLang versus standard Python and Manim. We implemented four algorithms representing di erent types of algorithms a user could use: binary tree inorder traversal (recursive, tree-based), Towers of Hanoi (recursive, stack-based), bubble sort (sorting, array-based) and memoised Fibonacci (dynamic programming, array-based). In addition to this we

considered both cases in which the user may or may not want to animate code tracking.

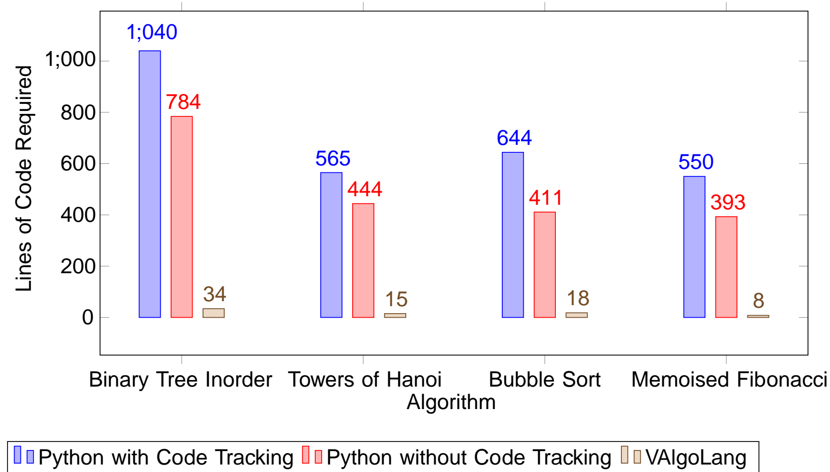

Figure 10: Lines of code required to animate algorithm in Python with Manim versus VAlgoLang

In Figure 10, it is evident that in all cases we see a signi cant decrease (at least 95%) in the lines of code required to produce the animation in VAlgoLang. This implies much less e ort is required to animate an algorithm in VAlgoLang than Python with Manim. This was achieved by heavily abstracting the animation construction from the user so they can focus on the algorithm itself.

Secondly, we explored the possibility that an educator may want to animate di erent cases of the same algorithm. This would have the purpose of potentially teaching algorithmic complexity such as inputs that achieve the algorithm's best and worst case.

We took the bubble sort algorithm and investigated, with increasing input array size, the additional or modi ed lines of code in VAlgoLang and Python with Manim required to animate the best / worst cases. This is relative to an animation of bubble sort with a singleton array as input.

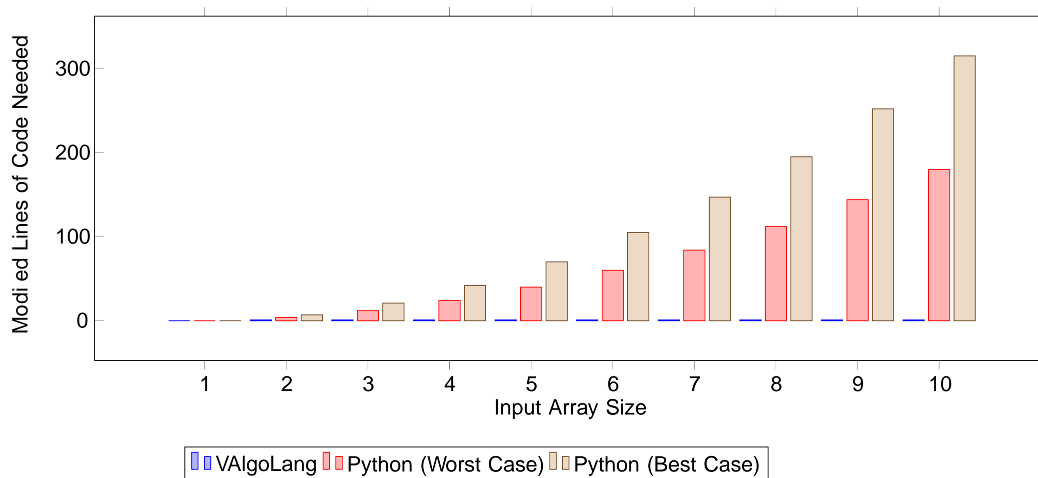

Figure 11: Chart showing the additional or modi ed lines of code required to animate Bubble Sort's best and worst cases with varying input size

As seen in Figure 11, there is a very steep increase in the code required to animate the change in input

for bubble sort Python and Manim. This includes animating with and without code tracking whereas in VAlgoLang, one must only change one line which modi es the input array to have the same e ect.

3.3 Trade o

With the ease of use VAlgoLang provides, there comes a trade-o . It o ers \o -the-shelf" visualisations that are exible enough for general use, however lacks immediate ner control over the animation. We understood this early on in conversations with our supervisor and provided the option to generate the Python le - both in the CLI (see Section B of the Appendix) and in the web editor as a check box option. This solution provides a very powerful o ering for those who would like to customise their animation for a speci c purpose e.g. the Towers of Hanoi algorithm. This algorithm, when implemented in VAlgoLang, will quickly and easily produce a standard visualisation for a stack (tower), with elements being of equal width. At this point, the basic visualisation is done. However, with additional modi cations to the Python le, an expert user can adjust the representation of the stack to set the width of the elements in proportion to their values as well as modifying the colours of each element.

3.4 User Feedback

Interviewing potential users formed a central part of our development cycle. We valued detailed, face-to-face qualitative feedback over broad surveys as we felt this form of feedback allowed us to really understand the needs of potential users in greater depth.

With our supervisor (Tony Field) being a possible future user, we were able to collaborate closely with him throughout the project checkpoints. We met with him on a regular basis, sometimes more than once between checkpoints, to gain his insight on what a lecturer wants from a tool like this. We directly applied what we took from these meetings to our iteration planning. One example of this is how based on Tony's feedback we removed all styling from the VAlgoLang syntax, leading to the creation of the stylesheet.

In addition, we conducted interviews with Undergraduate Teaching Assistants at Imperial College London who teach algorithms and data structures regularly. They are an important potential clientele for this project and with their dual experience as recent programming beginners and now teachers, so the sessions proved invaluable. We conducted these by rst asking how they prepare for their teaching sessions and what tools, if any, they use. These questions quickly revealed that the tool had to be quick and easy to use given the time constraints they are under. They were delighted to see how VAlgoLang ended up meeting their needs in this regard. Through these interviews another use case for VAlgoLang was discovered; independent learners could bene t from visualising solutions to algorithmic problems, such as on LeetCode, for Software Engineering interview practice.

We also met with other lecturers in the Department of Computing at Imperial teaching courses targeted at earlier years that involve a substantial component of data structures and algorithms. Mid-development of VAlgoLang we met with Robert Chatley where we spent some time discussing what he currently does to visualise algorithms. He explained how he uses slides in the Kotlin introduction course for rst year students and where VAlgoLang could t in.

Towards the end of the project, we met with Nicholas Wu, who lectures algorithms to second year students at Imperial. Nick enjoyed hearing about the technology behind the project. However, he felt that it might have been a better choice to implement an existing programming language like Haskell. Producing di erent frontend syntactic and semantic bindings to give the user control over the programming language they would like to use is something we would explore in the future.

Finally, we spoke to Alastair Donaldson, who lectures a large part of the Java module to rst year students. He was very enthusiastic and suggested it might be a great solution for certain students, if not a \one size ts all" approach. He was also supportive of our choice to develop our own pseudo-code like programming

language to keep things focused on the algorithms themselves.

We also engaged with the community of educators and animators online. Our project is now open-source and has, in a relatively short amount of time, accumulated 17 stars on GitHub, re ecting some wider interest online.

3.5 Limitation: Manim Processing Time

Early on in development, we realised that a clear limitation was the time needed for Manim to generate the video le. As shown in Table 1, on average, Manim takes more than 99% of the processing time for VAlgoLang from input le to video output. This is measured from running various algorithms multiple times and taking the average of the time taken at each stage.

Stage	Proportion of Average Time Taken (%)
Lexer & Parser	0.124
AST	0.0838
Virtual Machine	0.509
Python Writer	0.0224
Manim Animation Generation	99.3

Table 1: Table Showing The Average Processing Time Percentage Breakdown By Stages

3.6 Future Work

This project was initially an explorative task to discover what options there are to address the complexities of Manim. After these few iterations, the group felt like we had made a step in the right direction and would like to continue to do so. This feeling was driven by praise from our supervisor, users we had interviewed and the Manim community. That is why, throughout the development, we have ensured to design our code to be extensible with the opportunity to make this project open source. We would like to see the following implemented further:

- A more powerful and generalised language. This would be done by adding features such as user de ned structs/classes, further control structures (whenor switch clauses).
- Further data structures such as hash maps, linked lists, directed graphs.
- User-de ned syntax bindings to enable a user to choose the language used.
- Building an open source community of contributors. We would love to actively participate in a community of developers who would like to see this tool become more powerful.
- Developing further language support o ine by implementing plugins for IDEs which will allow for linting and syntax highlighting.

4 Ethical Considerations

4.1 License/Copyright

The interpreter part of the project is open-source under the BSD 3-Clause license [19], as the software is intended to be as freely available and open to contributions as possible while the licenses for the dependencies are taken into account. Furthermore, to ensure a welcoming and harassment-free environment for everyone, a code of conduct (adapted from the Contributor Covenant [20], which is commonly used in the open-source community) and contributing guidelines have been added. Developers should follow these if they wish to contribute.

In addition, the frontend and backend of the online editor are developed with dependencies under permissive license, which allows free development of the software. For functions referenced from other sources, they have been clearly stated in the code.

4.2 Accessibility

Subtitles are supported in the videos - users can simply add them with a@subtitle annotation, which allows them to add explanations of the algorithm during execution, catering for audiences who may have trouble hearing a narration over the video someone may add. The placement manager also allows users to create data structures larger and smaller, including the code block, which means parts of the animation can be made larger to cater for visually impaired.

The default colours used to generate the animation have also been carefully chosen. They have been veri ed to have high enough contrast ratio with the background for users to view, including those with visual disabilities.

4.3 Protection from malicious users

Malicious users may attempt to overload the backend API with many requests in a short period of time. While the impact may be low due to the chosen serverless architecture, the API backend server has a limiter on it, which tracks IP addresses and makes sure a single IP address does not send API requests more than a certain limit, preventing malicious users overloading the API.

5 Conclusion

We believe that we worked hard for each checkpoint to accomplish the goals of the project. Initially, we set out to simplify the process of creating educational animations for Computer Science with Manim. VAlgoLang achieves this by allowing the user to focus on the algorithm implementation rather than animation code. Additionally, we aimed to provide users with ne control over the animation - this is accomplished by the stylesheet, annotations and web editor features. We believe VAlgoLang has become a useful, user-friendly platform for Computer Science educators to create animations for data structures and algorithms e ectively, and hope it will continue to grow alongside our community.

References

[1] GitHub User bxff. Proper Documentation; 2020. Available from: https://github.com/3b1b/manim/issues/983.

[2] Neal D. Euler Tour Docs Page; 2021. Available from: https://eulertour.com/docs/.

[3] Bobek E, Tversky B. Creating visual explanations improves learning. Cognitive Research: Principles and Implications. 2016;1(27).

[4] Sanderson G. 3b1b/manim; 2021. Available from: https://github.com/3b1b/manim.

[5] Sanderson G. 3Blue1Brown YouTube Channel; 2021. Available from: https://www.youtube.com/channel/UCYO_jab_esuFRV4b17AJtAw.

[6] Sanderson G. 3Blue1Brown FAQ Page;. Available from: https://www.3blue1brown.com/faq#manim.

[7] KDoc; 2021. Available from: https://kotlinlang.org/docs/reference/kotlin-doc.html.

[8] Kotlin; 2021. Available from: https://kotlinlang.org/.

[9] ANTLR; 2014. Available from: https://www.antlr.org/index.html.

[10] React; 2021. Available from: https://reactjs.org/.

[11] Express; 2021. Available from: https://expressjs.com.

[12] TypeScript; 2021. Available from: https://www.typescriptlang.org/.

[13] Monaco Editor; 2020. Available from: https://microsoft.github.io/monaco-editor/.

[14] Docker Hub; 2021. Available from: https://hub.docker.com/.

[15] Netlify; 2021. Available from: https://www.netlify.com/.

[16] Docker; 2021. Available from: https://www.docker.com/.

[17] Cloud Run; 2021. Available from: https://cloud.google.com/run.

[18] Jest; 2021. Available from: https://jestjs.io.

[19] Initiative OS. The 3-Clause BSD License;. Available from: https://opensource.org/licenses/BSD-3-Clause.

[20] Ehmke CA. Contributor Covenant Code of Conduct;. Available from: https://www.contributor-covenant.org/version/1/4/code-of-conduct/.

Appendix A VAlgoLang Towers of Hanoi Example

Figure 12: Towers of Hanoi using VAlgoLang (view here)

Appendix B CLI

```
Usage: valgolang [-fhmpV] [--preview] [--progress_bars] [-o=<output>]
                 [-q=<quality>] [-s=<stylesheet>] <file>
VAlgoLang interpreter to produce manim animations.
      <file>                The .val file to compile and animate.
  -f, --open_file           Show the output file in file manager (optional).
  -h, --help                Show this help message and exit.
  -m, --manim               Only output generated python & manim code
                              (optional).
  -o, --output=<output>     The animated mp4 file location (default: out.mp4).
  -p, --python              Output generated python & manim code (optional).
      --preview             Automatically open the saved file once its done
                              (optional).
      --progress_bars       Print out and leave progress bars from manim
  -q, --quality=<quality>   Quality of animation. [low, medium, high] (default:
                              low).
  -s, --stylesheet=<stylesheet>
                            The JSON stylesheet associated with your code
  -V, --version             Print version information and exit.
```

Appendix C Ease of use metrics raw data

Code for algorithms used: https://github.com/VAlgoLang/VAlgoLang/tree/master/examples

Algorithm	Python Manim		VAlgoLang		Percentage Decrease (%)	
	With Code Tracking	Without Code Tracking	With Code Tracking	Without Code Tracking	With Code Tracking	Without Code Tracking
Binary Tree Inorder Traversal	1040	784	34	34	96.73	95.66
Towers of Hanoi	565	444	15	15	97.35	96.62
Bubble Sort	644	411	18	18	97.20	95.62
Memoised Fibonacci	550	393	8	8	98.55	97.96

Table 2: Lines of code required to animate algorithm in Python with Manim versus VAlgoLang

Input Size	Number of Additional or Modi ed Lines Required to Animate Bubble Sort			
	VAlgoLang Best Case	VAlgoLang Worst Case	Python + Manim Best Case	Python + Manim Worst Case
1	0	0	0	0
2	1	1	7	4
3	1	1	21	12
4	1	1	42	24
5	1	1	70	40
6	1	1	105	60
7	1	1	147	84
8	1	1	195	112
9	1	1	252	144
10	1	1	315	180

Table 3: Additional or modified lines of code required to animate Bubble sort's best and worst cases with varying input size

Appendix D Manim limitation raw data

Conditions: animations of low quality, no code tracking

System: Mid 2015 MacBook Pro, 2.2 GHz Quad-Core Intel Core i7, 16 GB 1600 MHz DDR3 Memory

Code for algorithms used: https://github.com/VAlgoLang/VAlgoLang/tree/master/examples

Stage	Time Taken (ms)												
	Binary Tree Inorder Traversal			Bubble Sort			Memoised Fibonacci			Towers of Hanoi			Average
	1	2	3	1	2	3	1	2	3	1	2	3	
Lexer & Parser	136	142	138	125	132	128	105	108	106	104	105	105	122.4
AST	109	112	108	80	95	87	54	58	55	67	68	67	82.5
Virtual Machine	470	453	485	523	543	557	495	511	508	465	441	459	501
Python Writer	20	18	23	22	31	25	18	19	18	27	31	29	22.1
Manim Animation Generation	133106	133032	134304	94029	94323	94194	38337	36343	37339	182460	182472	182342	97746.7

Table 4: Average processing time per stage in the VAlgoLang interpreter