A CPSC 647 PROJECT

Feynman-Path Simulation and Visualization

Project ID: 06

Aidan Evans
Department of Computer Science
Yale University
aidan.evans@yale.edu

Zhiyao Ma
Department of Computer Science
Yale University
zhiyao.ma@yale.edu

December 10, 2021

Prof. Yongshan Ding
Project Advisor

Abstract

Our project runs parallel Feynman-path simulation targeting a single server or serverless architecture and visualize the calculation path in hopes of providing researchers a tool to help understand the precise role of interference in quantum circuits. We mathematically describe how we turn Feynman-path simulation into a parallel algorithm. We describe our implementation in C++ targeting the two architectures. The visualization of the calculation path consists of a graph-based diagram showing the intermediate amplitudes of each possible state throughout the computation. Color-coded edges of various thickness will show the sign and magnitude of each amplitude and nodes will represent each possible state. Nodes will be displayed in a layered structure with each layer containing states possible for a specific step in the computation.

1 Introduction

Our project runs a parallel version of Feynman-path simulation and visualize the computation path. Users of our application can manipulate the quantum circuit, the input state and the output state graphically in our frontend. The user input will be sent to the simulation back-end through a web API, where the simulation is actually performed. After the result is returned, the frontend will render the intermediate results and weights along the computation path. Currently, the only tool to our knowledge which currently constructs visualizations of the Feynman-path relies on the circuit being constructed in Python. This tool also does not display as much information as it could; while it does change the arrow color based on the amplitude's sign, it could provide more information on the exact transitions between states and make reading the diagram more intuitive. Our implementation includes these additional pieces of information that previous versions of a Feynman-path tool do not display and make the creation of the diagram easier than having to program it through Python.

We first give a formal mathematical algorithm description of the parallel Feynman-path simulation. We break each intermediate state into sets and view the gates as mappings operating on sets. Each thread can then run in parallel and independently work on a set. With the algorithm formulated, we describe our implementation with C++. A straightforward implementation is based on the standard C++ asynchronous library where the compiled simulation program is running in a single process with multiple threads.

Additionally, we choose to target serverless architecture to run the simulation because we consider it to be the best way for personal user or small enterprise to perform classical simulation. Serverless completely relieves the user from managing servers and users will only be charged at the time their submitted computation request is running. It allows high concurrency, which easily goes beyond thousand, so we can achieve low task completion time way below that on a personal workstation. We make considerable engineering effort to break the algorithm into lambda handlers, provide indirect data passing through external key-value storage, and establish control channels.

We also provide a frontend interface for the generation of Feynman-path diagrams and easy interaction with the backend. We provide various configuration options for the diagram and the ability to dynamically interact with the generated diagram or download it as an image. The diagram itself clearly displays how the signs and magnitudes of both the real and imaginary parts of the computation paths change throughout the computation. The path itself is drawn using curved edges so that the edges between states reflects that of a natural flow. From testing, we have determined that the frontend can easily handle computations with upwards of 2000 unique paths.

2 Backend Simulation

In this section we introduce the design and implementation of the backend, where the description of gates are sent and the state of computational stage is generated.

2.1 Algorithm Design

We run Feynman path algorithm in the backend. Here we give a brief introduction of iterative Feynman path simulation algorithm.

2.1.1 Sequential Feynman Path Algorithm

The input of the algorithm is a list of gates. Denote the length of the list as s. In our implementation we restrict the accepted type of gates to a chosen set of single and double qubit gates. For single qubit gates, we accept Pauli-X, Y and Z gate, S gate (rotation of $\pi/2$ along the z-axis), T gate (rotation of $\pi/4$ along the z-axis) and Hadamard gate. For double qubit gates, we accept the controlled version of all of the single qubit gates above. These gates are universal.

Additional inputs are the number of qubits and whether we need to calculate the associated amplitudes of each intermediate states.

The algorithm starts from the initial state $|\psi_0\rangle = |0\rangle^{\otimes n}$. For each gate in the input list, denoting the gate as a unitary matrix U_i, the iterative way of calculating the state after applying the i-th gate is

$$|\psi_i\rangle = U_i |\psi_{i-1}\rangle . \tag{1}$$

The algorithm output is the set of the intermediate states and the final state $\{|\psi_1\rangle , |\psi_2\rangle , \cdots , |\psi_s\rangle\}$.

However, calculating the states by performing matrix-vector multiplication as in Eq.(1) has high performance overhead. The observation is that by our chosen gate set, each U_i is sparse and has a relatively simple structure. We thus directly consider the mapping effect of each unitary matrix.

The first step is to change the internal representation of a state from a vector to a set. Each element in the set is the pair of a basis state and the associated amplitude.

$$|\psi_{i-1}\rangle := \{\alpha_{i-1}^{(0)} |0\cdots00\rangle , \alpha_{i-1}^{(1)} |0\cdots01\rangle , \alpha_{i-1}^{(2)} |0\cdots10\rangle , \cdots , \alpha_{i-1}^{(2^n-1)} |1\cdots11\rangle\} \tag{2}$$

Given a gate, its effect when acting on a basis is to map it to another basis and to multiply a coefficient to the amplitude. Take Pauli-Y gate acting on the 0-th qubit ($Y^{[0]}$) as an example.

$$Y^{[0]}(\alpha |x0\rangle) = -i \cdot \alpha |x1\rangle \tag{3}$$

$$Y^{[0]}(\beta |y1\rangle) = i \cdot \beta |y0\rangle \tag{4}$$

Here x and y are arbitrary binary strings of length $(n-1)$. Applying a gate G on a state is equivalent to applying the map to all of the elements in the set to form a new set.

$$|\psi_i\rangle = G(|\psi_{i-1}\rangle) := \{G(e) | e \in |\psi_{i-1}\rangle\}. \tag{5}$$

For the gate set we accept, the time complexity for mapping a single element (i.e. a single basis with its amplitude) is $O(1)$. Thus, assuming that we run the algorithm sequentially, the total time complexity is the sum of the cardinal number of the intermediate state sets

$$O(\sum_{1}^{s} Card(|\psi_i\rangle)). \tag{6}$$

2.1.2 Parallel Feynman Path Algorithm

Now we parallel the execution of Feynman path algorithm. Observe that the mapping procedure of each element in a set is independent from each other, so we can split the mapping into several concurrent running threads. Mathematically,

$$G(|\psi_i\rangle) = G(|\psi_i^1\rangle) \cup G(|\psi_i^2\rangle) \cup \cdots \cup G(|\psi_i^t\rangle), \text{ where } |\psi_i\rangle = |\psi_i^1\rangle \cup |\psi_i^2\rangle \cup \cdots \cup |\psi_i^t\rangle . \tag{7}$$

Each thread processes a "partial state" (e.g. $|\psi_i^t\rangle$). When all of the running threads finish a stage, we can then aggregate the partial results together to get back the intermediate state. Note that we do not need to aggregate immediately after each stage. Threads can continue to apply the following gate. We can even choose to aggregate back the states when we finish all of the stages.

For the sake of simplicity, above we slightly abused the mathematical notation. Not all of the gates we accept in our algorithm is one-to-one mapping, specifically the Hadamard gate. In this case the mapping is one-to-two.

$$H^{[0]}(\{\alpha |x0\rangle\}) = \left\{ \frac{\alpha}{\sqrt{2}} |x0\rangle , \frac{\alpha}{\sqrt{2}} |x1\rangle \right\} \tag{8}$$

$$H^{[0]}(\{\alpha |y1\rangle\}) = \left\{ \frac{\alpha}{\sqrt{2}} |y0\rangle , \frac{-\alpha}{\sqrt{2}} |y1\rangle \right\} \tag{9}$$

Hadamard gate leads to the growth of the set. For instance if the t-th thread is working on $|\psi_{i-1}^t\rangle$ and at the current stage it should apply a Hadamard gate (e.g. on qubit 0). The new set after mapping will have double cardinal number than the previous one.

$$Card(|\psi_i^t\rangle) = Card(H^{[0]}(|\psi_{i-1}^t\rangle) = 2 \cdot Card(|\psi_{i-1}^t\rangle)) \tag{10}$$

Thus, to keep the working set of each thread within a manageable size, when it reaches a size threshold T, we partition it into two sets of equal size and spawn a new thread to work on the new one.

2.2 Implementation

We implemented two versions of the backend server. They provide an identical interface facing the frontend. The whole code base is over 2000 lines of C++ code. Most of the code is shared between the two versions, but the control channel, intermediate state storage and asynchronous function invocation parts are architecture specific. A compilation flag controls which architecture to target.

The backend exposes an HTTP API. Requests to the backend are sent through HTTP POST. Each request should contain the number of qubits, a list of gates, and whether or not the amplitude is needed in the result. Request arguments should be packed in JSON format sent through the HTTP POST payload. The backend sends the result back through the HTTP response, containing

the basis and amplitudes of each intermediate state, also packed in JSON format. The backend implements the HTTP server with uWebSockets[1].

When indicated that amplitude is not required in the result, the backend responds with the basis that has non-zero amplitudes at each stage, but does not send back the amplitude, not even calculate them during the computation.

Note that although in section 2.1.2 we explore the possibility that iterating through subsequent gates with partial states can happen concurrently with aggregating calculated partial states, we do set additional synchronization barriers in the implementation. The reason is merely to limit the number of threads running in parallel. Otherwise, for an large enough input, we may exhaust system resource.

Below we discuss the two implemented deployment architecture.

2.2.1 Single Server Deployment

In single server deployment we ran the backend as a single process on a multi-core machine. Since all of the working threads live in the same address space, they can pass data to each other by exchanging pointers. The states can always be stored in memory and need not to be written out to external storage.

Threads in single server deployment are spawned through the standard C++ asynchronous API (i.e. `std::async`). The API returns a `std::future` object which can be used as the join handle of the spawned thread. When all of the threads calculating for the partial states of stage i finishes, we join back all of them. Next, a new thread is spawned to aggregate partial results

$$|\psi_i\rangle = Aggr(|\psi_i^1\rangle, |\psi_i^2\rangle, \cdots, |\psi_i^t\rangle).$$

At the same time, t threads are spawned to continue to process $|\psi_i^r\rangle, r \in \{1, 2, \cdots, t\}$.

A thread will output two pointers by splitting the result set if after the mapping its working set exceeds a predefined size threshold, but normally it will output only one pointer.

The controller thread waits for all of the newly spawned threads to finish before proceeding to the next iteration, to limit the number of concurrent threads.

2.2.2 Serverless Deployment

Function as a service (FaaS) is now a growing workload in the cloud. Service providers allow their customer to upload programs in source files written in high level language or as compiled executable binaries. These programs will be invoked when associated condition is met. For instance, a trigger can be set to monitor whether a new file is uploaded to a cloud storage and invokes the function automatically when it happens. The uploaded programs are usually named lambda handlers. Additionally, programmers can invoke the uploaded program directly through the cloud provider's SDK. Developers are freed from maintaining the server and managing the infrastructure. They can simply get their computation task done when it needs to be done, thus the name serverless.

From the customer side, the merit of moving to serverless is that the cloud provider can provide a concurrency level counting in thousands. Moreover, cloud providers only bill their user based on actual usage. That means when the program is not running, nothing is charged to the user.

[1] https://github.com/uNetworking/uWebSockets

We believe that the workload of classical simulation is highly spiky. Most of the time we are idle, but when we are performing the simulation we need extraordinary high concurrency. The pattern matches well with what serverless provides.

We choose Amazon AWS Lambda[2] as the FaaS provider, because it is publicly available and provides free computation time. The maximum concurrency we can obtain from it is 1000[3]. Each running instance can receive at most 10GB memory[4].

The control logic of the serverless deployment is the same as in the single server one, but the implementation of data passing and thread spawning are different. Each thread in the single server deployment is now a separate running lambda handler instance. They are spawned in different containers and very often running on different machines. We establish a Redis key-value storage server to mediate data passing. Intermediate partial states are stored in this storage server and retrieved later.

The controlling thread in the serverless deployment is a process running inside an EC2[5] virtual machine instance. We establish publisher-subscriber channels through the Redis server to allow control information exchange between the controller and the worker lambda instances. Specifically, after a lambda instance writes its output to the Redis key-value storage, it reports back to the controller the keys it just stored through its channel.

All lambda instances are spawned by the controller through AWS SDK[6] API call. From the perspective of the spawned lambda, it is triggered by the direct invocation event. In the event payload sent from the controller to the spawned lambda, we embed the information it needs to perform its task and communicate with the controller. The embedded information includes the keys into the Redis key-value storage from which it retrieves its input partial states, the gate it should apply and the name of the channel it should use to communicate with the controller.

2.3 Discussion and Future Work

2.3.1 Exact Performance

Though it would be very informative to see how many qubits the backend can simulate given a realistic application, but since we are using Amazon AWS Free Tier[7], our monthly lambda computation time is limited. Moreover, the Redis server we deployed has very limited bandwidth and storage quota. What we have demonstrated is the feasibility to run Feynman path simulation workload with serverless architecture, and its potential is promising.

2.3.2 Partial State Merging

One loose end in our algorithm implementation is that partial state sets can be no longer "disjoint" after passing through a Hadamard gate. As a minimal example, consider that two threads are

[2] https://aws.amazon.com/lambda/

[3] https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html

[4] https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html

[5] https://aws.amazon.com/pm/ec2/

[6] https://aws.amazon.com/sdk-for-cpp/

[7] https://aws.amazon.com/free/

concurrently applying a Hadamard gate to their respective partial set $|\psi_1\rangle$ and $|\psi_2\rangle$. Suppose that

$$|\psi_1\rangle = \{\alpha\,|0\rangle\}, \tag{11}$$

$$|\psi_2\rangle = \{\beta\,|1\rangle\}. \tag{12}$$

After applying a Hadamard gate, the new partial states $|\psi_1'\rangle$ and $|\psi_2'\rangle$ are not "disjoint" because they share common basis.

$$|\psi_1'\rangle = H(|\psi_1\rangle) = \left\{\frac{\alpha}{\sqrt{2}}\,|0\rangle\,, \frac{\alpha}{\sqrt{2}}\,|1\rangle\right\} \tag{13}$$

$$|\psi_2'\rangle = H(|\psi_2\rangle) = \left\{\frac{\beta}{\sqrt{2}}\,|0\rangle\,, \frac{-\beta}{\sqrt{2}}\,|1\rangle\right\} \tag{14}$$

When common basis occurs between two sets, the speed up gained from multi-threading is not linear anymore. In the worst case, all of the threads might work on exactly the same set of basis, yielding no speed up. Thus, we should merge together two sets when they share too many common basis. However, this is yet to be implemented and we leave it as a future work.

One challenge of implementing the merge algorithm is to decide which two sets need to be merged. Simply counting the number of basis they share is intractable because it requires calculating the intersection between each pair of the sets. One possible approach is to sample from each set and then observe how similar the samples are between pairs. The more the samples looks similar, the likely that two sets share many common basis.

3 Frontend Visualization

3.1 Related Work

A previous Feynman-path diagram creation tool[8] exists but this tool does not display as much information as it could and sometimes is not the easiest to read. We used this tool as a starting point in how the generated diagram should look; this fact can be seen in our choice of colors for the sign of the real-valued amplitudes, discussed below: blue for positive, orange for negative.

3.2 Method

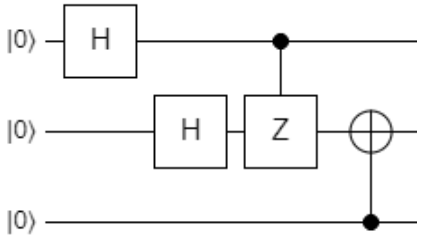
3.2.1 Input Format

We specify the input circuit by uploading a JSON file containing the circuit via a file upload button on the frontend. The JSON is of the same format as that exported by the popular quantum circuit simulator Quirk[9]. Therefore, the exported JSON from Quirk can be easily uploaded and run on the frontend.

The specific JSON format is described as follows: The JSON contains one key/value pair where the key has the field name "cols" and the value is an array of arrays. Each array in the main array specifies the gates to be applied during a stage of the computation. Gates in each stage are denoted

[8] https://github.com/cduck/feynman_path

[9] https://algassert.com/quirk




```
1: {
2:     “cols”: [
3:         [“H”],
4:         [1, “H”],
5:         [“•”, “Z”],
6:         [1, “X”, “•”]
7:     ]
8: }
```

Figure 1: Example JSON and corresponding circuit.

by a corresponding character at the index of the array associated with the qubit the gate is to be applied to; for example, if an “H” is at the i^{th} index of the array, then at this stage, a Hadamard gate is applied to qubit q_i.

Further formatting specifics are best explained with an example. Consider the JSON depicted in figure 1. Per the above reasoning, the array on line 3, applies a Hadamard gate to qubit q_0. On lines 4 and 6, the ones represent the identity gate and essentially act as padding so that later gates may be properly indexed by the qubit they apply to. Accordingly, on line 4, a Hadamard is applied to qubit q_1. On lines 5 and 6, “•” denote that this stage of the computation applies a controlled gate. Thus, line 5 represents a controlled-Z gate with qubit q_0 as the control and q_1 as the target; similarly, line 6 represents a controlled-X gate with qubit q_2 as the control and q_1 as the target.

3.2.2 Backend Communication

Before we send the circuit JSON to the backend to perform the simulation, we first preprocess the circuit in order to standardize the types used in the array and convert complicated UTF-8 characters into simpler ones. This surmounts to changing all integer ‘1’s to the string “I” and changing all “•”s to “.”s.

We then add a “qubits” field to the JSON with the number of qubits used in the circuit and and depending on what specific settings are desired for simulation, a boolean “with_amp” field – the “with_amp” field is further explained in section 3.2.3.

Once the circuit JSON is fully preprocessed, it is then sent to the backed by performing an HTTP POST request using the JavaScript Fetch API. At the moment, since we run the frontend locally and the backend on a separate domain, in order to circumvent CORS policy, a proxy server[10] is run locally for making the request to the backend.

3.2.3 Diagram Generation

After making the HTTP POST request to the backend and receiving the simulation data, we then process this data to set up the Feynman-path diagram. We leverage the JavaScript library Konva[11] to generate the diagrams because of its efficient and easy-to-use interface. figure 2 depicts the

[10] https://github.com/nsnave/feynman-path-diagram/blob/master/test-server/cors.py

[11] https://konvajs.org/docs/index.html

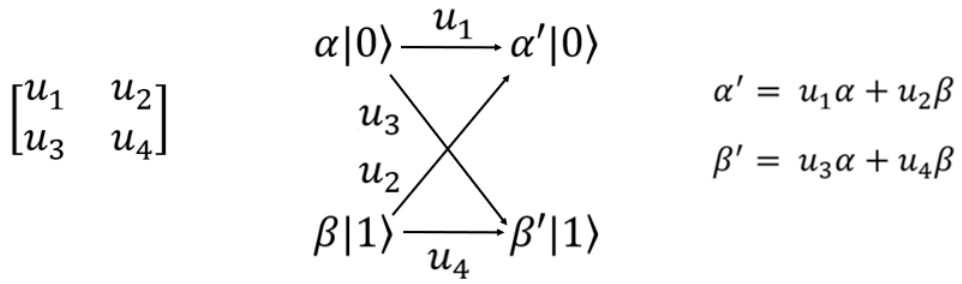


Figure 2: Diagram generation logic.

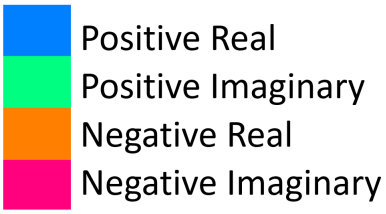


Figure 3: Arrow color scheme based on amplitude.

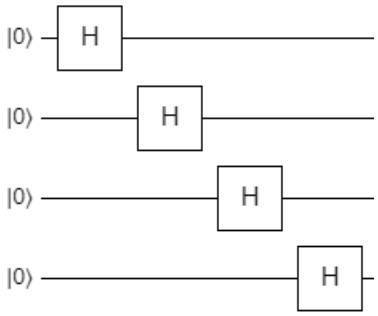


Figure 4: Example 1 Circuit

basic idea behind how the diagram is generated. Each row represents a possible state in the computational basis; each column represents a stage in the computation. The arrows between columns represent how the application of a unitary gate affects the state.

We apply this basic diagram generation process to generate diagrams that are both easy to read and convey useful information. We generate diagrams as a layered graph with nodes in each column corresponding to the possible state of the computation at that stage where. The radii of these nodes depend on the amplitude of the state; α, β, α', and β' in figure 2. Arrows between nodes corresponding to the specific part of the unitary – u_1, u_2, u_3, or u_4 – being applied where the width of these lines depend both on the unitary part and the amplitude of the node the arrow is originating from. For example, in figure 2, the width of the arrow between $\alpha\,|0\rangle$ and $\alpha'\,|0\rangle$ would depend on the values of u_1 and α. Specifically, the width is computed for an arrow associated with the unitary part u_i and originating from a node with amplitude α as follows:

$$\texttt{Weight}(\alpha, u_i) = |(\alpha u_i)(\alpha u_i)^*|$$

The color scheme of each node and arrow depend on the sign of the real and imaginary part of the amplitude or unitary part, respectively. Figure 3 depicts the colors associated with these signs. One may ask at this point: "What about numbers with both a real and imaginary part, such as $\frac{1}{2} + \frac{1}{2}i$?" The coloring scheme of these numbers uses both the real and imaginary colors combined together; further explanation on colors is provided with an example in section 3.3.7.

In addition to the feynman-path diagram itself, we also provide options for listing the gates being applied to each layer at the top of the diagram and the state associated with each row on the

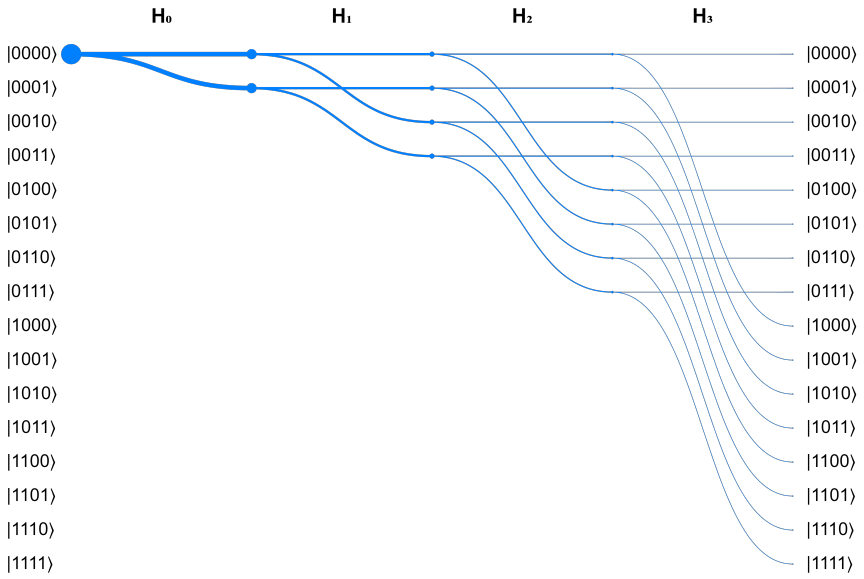


Figure 5: Example 1 Diagram

sides of the diagram; these are present in the examples below.

To avoid cluttering the diagram, some users may wish not to display the nodes and only care about the arrows; for this purpose, we provide an option to not display the nodes.

We also provide an option to permute the order in which the states are displayed. In the case of two qubits, this allows one to toggle between either ”00, 10, 01, 11” or ”00, 01, 10, 11” order. In some cases, this makes it easier to read the diagrams. We provide both the permuted and non-permuted diagrams for the examples in sections 3.3.1 and 3.3.2.

Finally, we also provide an option to not compute the amplitudes of the computation; the “with_amp” field mentioned previously is set to “False” in this case. This allows for the backend to compute the diagram faster; however, we lose meaningful information and, therefore, the diagram is no longer as useful. Not calculating the amplitudes also prevents us from knowing where interference *actually* happens and only where it *may* happen. We provide an example of this option in section 3.3.6.

3.3 Examples

We now discuss some examples to demonstrate the usefulness of our tool.

3.3.1 Example 1: Superposition

We first provide an example that demonstrates how the width of the nodes and edges changes based on the state amplitudes. The circuit, depicted in figure 4, consists simply of a Hadamard gate applied to each qubit. This causes the path to “split” at each stage of the computation, leading the node and arrow widths to grow smaller as the superposition of states increases. We include non-permuted and permuted variations of this diagram in figures 5 and 6, respectively.

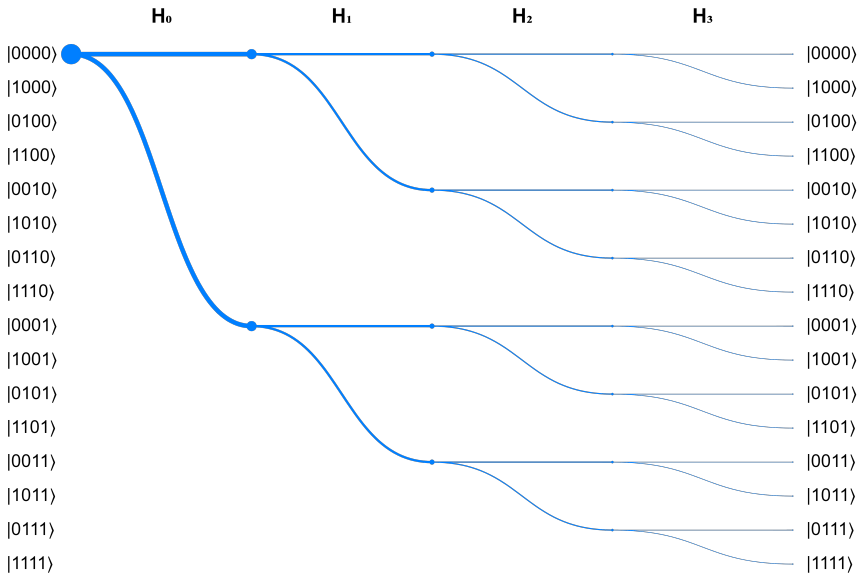


Figure 6: Example 1 Diagram Permuted

3.3.2 Example 2: Scalability

We further extend the circuit of section 3.3.1 to include Hadamard gates applied to ten qubits. This demonstrates that the diagram generation is scalable enough to handle tests with about ten qubits. Because the width of the arrows shrinks greatly at the later stages, the frontend interface allows users to zoom and move around the diagram. Because of the massive number of objects on the canvas, however, there is some lag when moving around the canvas but it was not enough to significantly impact use on ten and eleven qubit tests we tried. This is a case where toggling "All Nodes" to off may be useful since not adding the nodes significantly decreases the number of objects on the canvas. Section 5.2.1 in the Appendix contains the non-permuted and permuted diagrams for this example.

3.3.3 Example 3: Interference 1

One of the most usefull things Feynamn-path diagrams depict is interference. Consider the circuit in figure 7. This creates a Bell pair by taking advantage of destructive interference of the $|01\rangle$ and $|10\rangle$ states. Via the Feynman-path diagram we generated in figure 8, we can see this process happen. First look at the $|10\rangle$ row in the last layer of the computation; here we have a blue line and orange line converge. Since the blue represents a positive real and orange a negative real, they cancel out resulting in the amplitude of the $|01\rangle$ node in the last layer to equal zero – and, thus, is not displayed. Similarly, for the $|10\rangle$ row, we have two blue lines converge *but* one of the blue lines originates from an orange node and, therefore, the two lines which converge on $|01\rangle$ will still cancel out during the computation. We are then left with just the $|00\rangle$ and $|11\rangle$ nodes in the final layer.

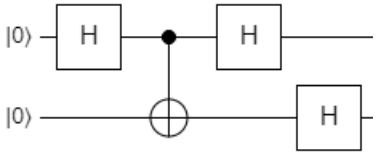


Figure 7: Example 3 Circuit

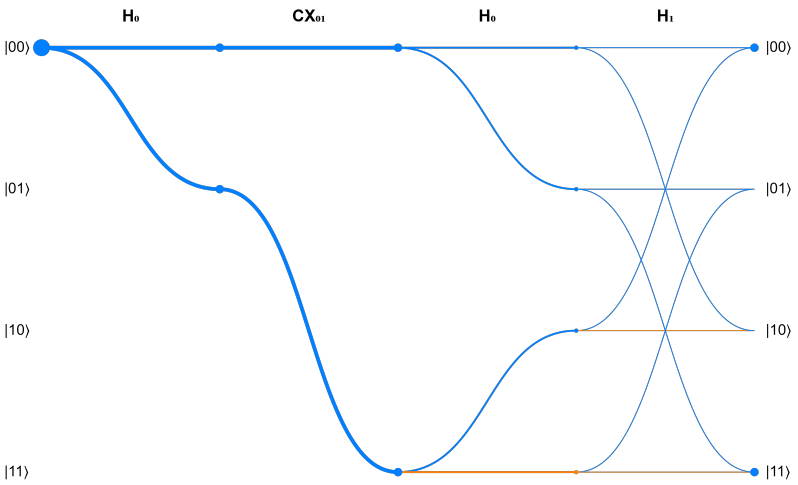


Figure 8: Example 3 Diagram

3.3.4 Example 4: Interference 2

Consider the circuit in figure 10 and its corresponding diagram in figure 9. With this circuit, we wish to point out the constructive interference which happens when the first Hadamard is applied to qubit q_1; i.e., the first "H_1" in the diagram. Here we see that the radius of the $|01\rangle$ and $|10\rangle$ nodes increase after H_1 due to constructive interference and the $|00\rangle$ and $|11\rangle$ nodes disappear due to destructive interference.

3.3.5 Example 5: Ancilla Introduction

As depicted in figure 11, we add a CNOT gate to copy the intermediate result of the Example 4 circuit onto an ancilla. This causes the interference which occurred in the previous example not to happen, showing why ancillae may affect the output of a computation. Notice that the corresponding diagram, figure 12, does not contain any interference during the first H_1 stage, as it did before adding the ancilla.

3.3.6 Example 6: Unchecking the "Get Amplitudes" Option

We now show two diagrams for the circuit in figure 15. Figure 13 depicts the circuit's diagram with amplitude calculation and figure 14 shows it without. Notice that because when constructive or destructive interference happens depends not only on the unitary but also the amplitudes, we cannot rule out the computation paths which destructively interfere with each other. Therefore, this explains why the computation path computed without the amplitudes includes more edges between nodes in the last layer than it did with amplitude calculation – we couldn't know whether destructive interference happened.

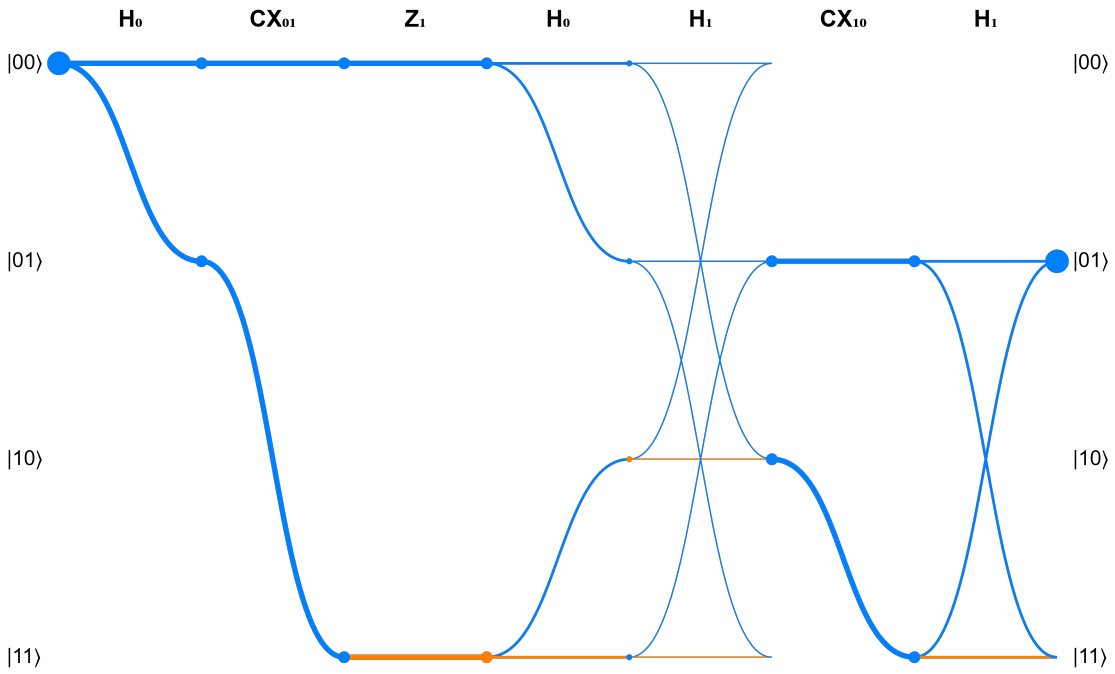


Figure 9: Example 4 Diagram

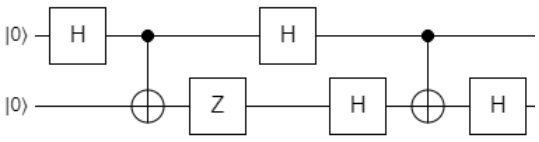


Figure 10: Example 4 Circuit

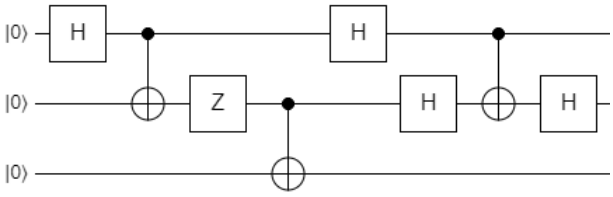


Figure 11: Example 5 Circuit

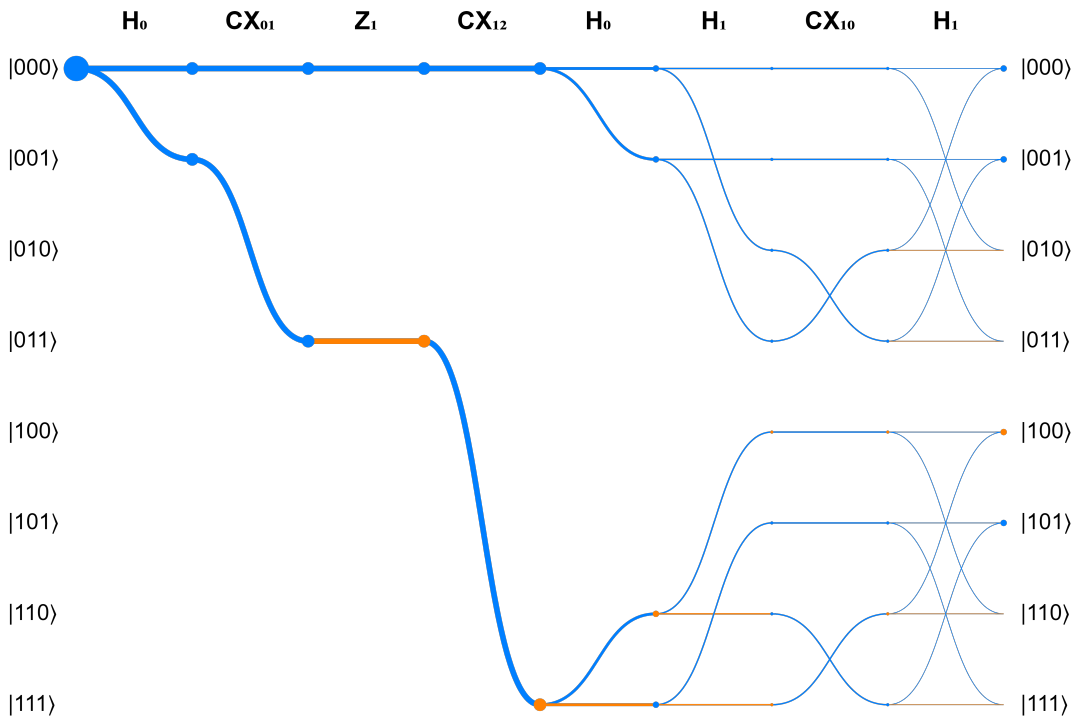


Figure 12: Example 5 Diagram

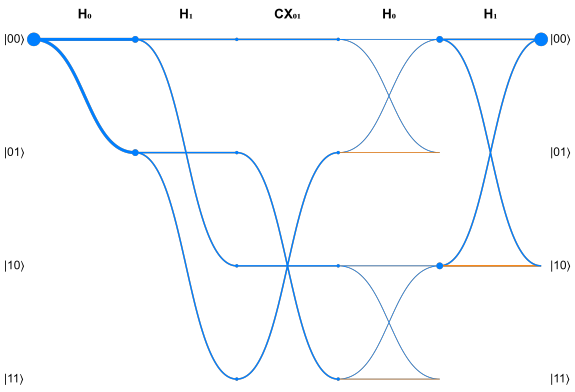


Figure 13: Example 6 Diagram With Amplitude Calculation

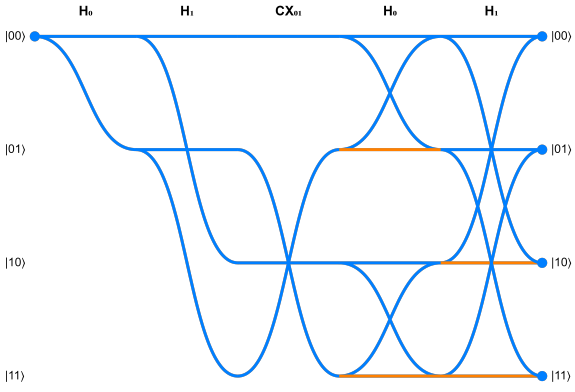


Figure 14: Example 6 Diagram Without Amplitude Calculation

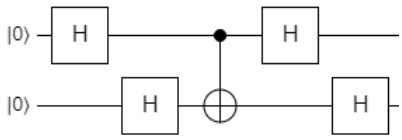


Figure 15: Example 6 Circuit

Figure 16: Example 7 Circuit

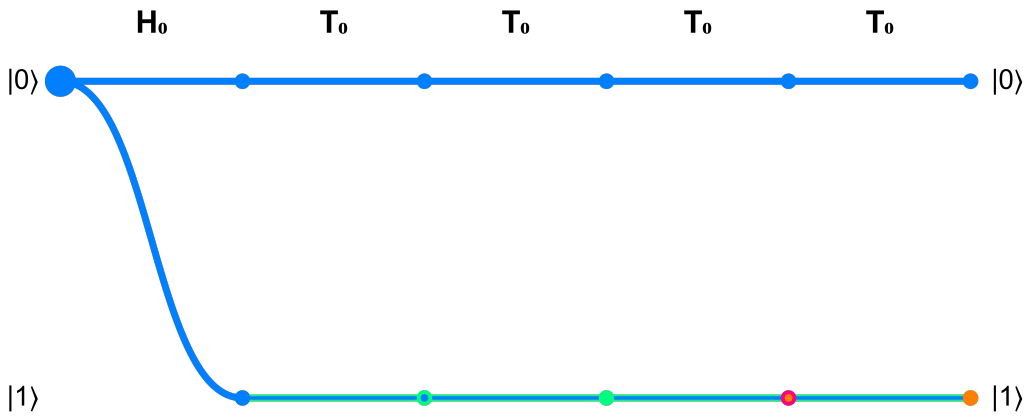


Figure 17: Example 7 Diagram

3.3.7 Example 7: Complex Amplitudes

Up to this point, we have dealt with circuits which only had real-valued amplitudes and unitaries. We now introduce a circuit with complex values. Consider the circuit in figure 16; this applies a Hadamard gate followed by four T gates. The corresponding diagram is depicted in figure 17. In the $|1\rangle$ row, each edge associated with a T gate is a mix of blue and green – the colors corresponding to positive real and positive imaginary values. This makes sense because a T gate is defined as

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{2} + \frac{1}{2}i \end{bmatrix} = \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \end{bmatrix}$$

Since we are looking at the edge from $|1\rangle$ to $|1\rangle$, the color-scheme of this edge will be based on $u_4 = \frac{1}{2} + \frac{1}{2}i$. For a complex number $a + bi$, the color ratio is determined as follows:

$$\texttt{Re} = \frac{|a|}{|a| + |b|}$$
$$\texttt{Im} = \frac{|b|}{|a| + |b|}$$

where `Re` is the percentage of the line which colored for the real part and `Im` for the imaginary part.

If the sign of a is positive the color of the real part is blue, else orange. If the sign of b is positive the color of the imaginary part is green, else pink. This is summarized in figure 3. Moreover, this choice of colors leads to positive values being associated with cooler colors and negative values being associated with warmer colors. The specific color hues were also chosen to achieve optimal contrast between any two adjacent colors.

Returning to the example, we can see how the amplitude changes after each T gate; the node color scheme is determined similarly to the lines'. As the T gates are applied, we see the real to

imaginary ratio of $|1\rangle$ amplitudes changes as follows:

$$(+100) : 0 \;\Rightarrow\; (+50) : (+50) \;\Rightarrow\; 0 : (+100) \;\Rightarrow\; (-50) : (-50) \;\Rightarrow\; (-100) : 0$$

This is in line with the actual amplitude changes of the $|1\rangle$ during the computation:

$$\frac{1}{\sqrt{2}} \Rightarrow \frac{1}{2} + \frac{1}{2}i \Rightarrow \frac{1}{\sqrt{2}}i \Rightarrow -\frac{1}{2} - \frac{1}{2}i \Rightarrow -\frac{1}{\sqrt{2}}$$

Moreover, we can also determine from this example that the output state is $|-\rangle$ because from just looking at the final nodes, we see that both have equal radii and the $|1\rangle$ amplitude is negative. Therefore, the final state will be

$$\frac{|0\rangle - |1\rangle}{\sqrt{(\langle 0| - \langle 1|)(|0\rangle - |1\rangle)}} = \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |-\rangle$$

which is what it should be since applying an H gate changes $|0\rangle$ to $|+\rangle$ and applying four T gates in a row is equivalent to applying a Z gate.

3.4 Future Work

The diagram could be made more interactive by having the exact numerical amplitude displayed when hovering over a node. We could also provide additional diagram options for the display of the specific amplitude values in the diagram itself. The circuit input method could also be improved by providing a text box for a circuit JSON to be pasted into, instead of uploading a file, or for a circuit creation tool to be embedded into the frontend so that both circuit creation and feynman-path simulation could exist in the interface.

4 Conclusion

In this project we implement a parallel version of Feynman-path simulation algorithm and deploy it on both a single server and serverless architecture. We provide an HTTP API interface allowing simulation requests to be sent to the backend with ease. We show that it is promising to run classical simulation in the cloud. We provide a frontend for easy communication with the backend and the generation of dynamic, intuitive diagrams displaying meaningful information about the computation. The frontend interface is available as a GitHub repository.[12].

[12] https://github.com/nsnave/feynman-path-diagram

5 Appendix

5.1 Backend Example

5.1.1 Backend API Example

```
import json
import requests

url = 'http://44.195.46.250:23333'

payload = {
    'qubits': 3,
    'with_amp': True,
    'cols': [
        ['H'],
        ['I', 'H'],
        ['I', 'I', 'S'],
        ['.', 'X'],
        ['Z', '.'],
        ['I', '.', 'Y'],
        ['I', 'T', '.'],
        ['.', 'I', 'H'],
    ]
}

malformatted_payload = {
    'qubits': 3,
    'with_amp': True,
    'cols': [
        ['R']
    ]
}

# Send HTTP post request and get the response.
mal_response = requests.post(url, json=malformatted_payload)

# Check the "success field".
mal_response_dict = json.loads(mal_response.text)
print('With malformatted input, "success" =', mal_response_dict['success'])

# Send HTTP post request and get the response.
response = requests.post(url, json=payload)

# Parse the response to a Python dict.
response_dict = json.loads(response.text)

# Make sure the invocation is successful.
assert response_dict['success']

print('Number of qubits:', response_dict['qubits'])
cnt = 0
for amplitude in response_dict['amplitudes']:
```

```
    # Print the amplitudes after each stage.
    print('Stage', cnt)
    for key, val in amplitude.items():

        # Print base and amplitude in the format "base  [real, imaginary]".
        print('%s\t [%.4f, %.4f]' % (key, val[0], val[1]))
    print()
    cnt += 1
```

5.1.2 Output Example

```
'''
With malformatted input, "success" = False

Number of qubits: 3
Stage 0
000      [0.7071, 0.0000]
001      [0.7071, 0.0000]

Stage 1
000      [0.5000, 0.0000]
010      [0.5000, 0.0000]
001      [0.5000, 0.0000]
011      [0.5000, 0.0000]

Stage 2
000      [0.5000, 0.0000]
010      [0.5000, 0.0000]
001      [0.5000, 0.0000]
011      [0.5000, 0.0000]

Stage 3
000      [0.5000, 0.0000]
010      [0.5000, 0.0000]
011      [0.5000, 0.0000]
001      [0.5000, 0.0000]

Stage 4
000      [0.5000, 0.0000]
010      [-0.5000, 0.0000]
011      [-0.5000, 0.0000]
001      [-0.5000, 0.0000]

Stage 5
000      [0.5000, 0.0000]
110      [0.0000, 0.5000]
111      [0.0000, 0.5000]
001      [-0.5000, 0.0000]

Stage 6
000      [0.5000, 0.0000]
```

```
110       [-0.3536, 0.3536]
111       [-0.3536, 0.3536]
001       [-0.5000, 0.0000]

Stage 7
000       [0.5000, 0.0000]
110       [-0.3536, 0.3536]
111       [0.2500, -0.2500]
011       [-0.2500, 0.2500]
001       [-0.3536, 0.0000]
101       [-0.3536, 0.0000]
'''
```

5.2 Frontend Diagrams

5.2.1 Example 2 Diagrams