Universiteit Leiden
The Netherlands

Opleiding Informatica

Quantum topological data analysis
on near-term devices

Marit Talsma

Supervisors:
Vedran Dunjko & Casper Gyurik

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

03/08/2021

Abstract

Topological data analysis (TDA) is an increasingly more investigated and used method in machine learning. The crux of TDA is the calculation of the number of high dimensional "holes" in a data set, which helps robustly characterise that data set. This process is computationally expensive, and it has been shown it can significantly benefit from the use of quantum computers. However, current quantum computers are limited in terms of the size and quality of their memory, limiting both the size and depth of the computation that can be run. In this thesis we seek to optimize the quantum topological data analysis algorithm for use on near-term quantum computers. The original quantum TDA algorithm works by relying on a certain class of algorithms for so-called Hamiltonian simulation, which is a step in the quantum TDA method. This class of algorithms leads to deep quantum computations and require many ancillary qubits. This approach is therefore not suitable for near-term quantum computers. In this thesis we will investigate the possibility of using the much more efficient, but limited, Trotterization-based algorithm for Hamiltonian simulation, which can lead to a more efficient scheme for quantum topological data analysis. In using this Troterrization-based approach, the key technical challenge is identifying a way to express the so-called combinatorial Laplacian matrices - which are key objects characterising the data in TDA - at least approximately in terms of as few as possible special matrices called Pauli strings. In this work we analyse the number of Pauli strings appearing in practice, and the robustness of the method of ignoring some of the terms to save computational resources. The presented research shows that up to 70 percent of these Pauli strings can be removed while keeping a good approximation for the resulting number of high-dimensional holes. Further we show that the error in this approximation can even be reduced using an additional trick we call the prefactor term. This work sets the basis for even more efficient quantum TDA algorithms which can be run on near-term quantum computers.

Contents

1 Introduction

1.1 Motivation

Topological data analysis [Mun17] is a data analysis technique that requires a substantial amount of computational power. Unfortunately classical computers are only strong enough to facilitate this for very small data sets. Quantum computers have the possibility of exponential speedups over classical computers for the topological data analysis problem, opening up the possibility of using topological data analysis on a large scale. Current quantum computers however are not yet able to be used to their full potential due to limitations in size and stability. Therefore in this thesis I investigate how to optimize the topological data analysis algorithm for running on near-term quantum computers.

The topological data analysis algorithm uses graphs to find high-dimensional holes in a data set. For the implementation of this algorithm on a quantum computer one can use Hamiltonian simulation based on Trotterization (which will be explained in Sections 2.2.4 and 2.2.5) or based on sparse access (Section 3). For a space efficient implementation we use Trotter-based Hamiltonian simulation and we will try to reduce the computational depth created by using this Trotterization method. During the process of reducing this computational depth an error term will be introduced which we will try to keep as small as possible.

1.2 Thesis overview

In this thesis I will give an overview of the inner workings of a quantum computer (Section 2.2), explain the topological data analysis technique (Section 2.1) and explain how topological data analysis can be performed on a quantum computer (Section 3). In Section 4 I explain the research question where I take a look at the possibility of implementing the space efficient Trotterization technique needed to run the quantum topological data analysis algorithm on near-term quantum computers, while keeping it as computationally inexpensive as possible. I have written code that calculates the combinatorial Laplacian from a graph and which can decompose this matrix into Pauli strings as explained in Section 5. In Section 6 I investigate the scaling of the number of Pauli strings in the graph size and how many, if any, Pauli strings can be discarded from the combinatorial Laplacian to reduce the computational depth while keeping the result within a certain error margin. These experiments show promising results for the reduction of the computational depth by discarding a large percentage of nonzero Pauli strings. Later these experiments are analyzed to investigate possibilities for further research (see Section 7).

2 Background

2.1 Topological data analysis

Topological data analysis uses topology to analyze data. The key step and also the computationally most expensive step of this procedure is finding the number of k-dimensional holes, or the k^{th} Betti number, of a given data set. Overall this key step consists of four parts, namely creating a

clique complex from the given data set, constructing the boundary map from this clique complex, computing the combinatorial Laplacian from the boundary map and calculating the desired Betti number. An overview of the process of topological data analysis can be found in Figure 1. In this section I will walk trough this technique step by step, explaining the concepts needed for the implementation of the quantum topological data analysis algorithm later in this thesis.

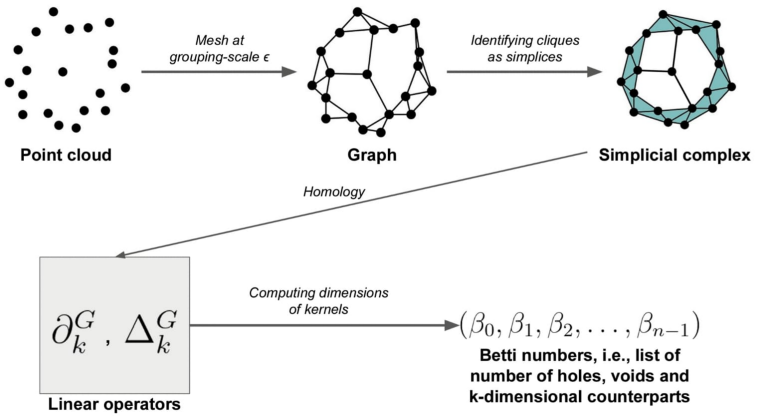


Figure 1: Topological data analysis pipeline [GCD20]

2.1.1 From data set to clique complex

The topological data analysis technique for calculating the k^{th} Betti number starts with the creation of a graph from a chosen data set. Between all data points an idea of proximity can be created, connecting certain data points which lie within a certain distance of each other using a chosen threshold. To define this threshold a grouping scale ϵ is used, where all nodes closer to each other than this grouping scale ϵ will be connected to each other. This results in the required graph. In topological data analysis a specific type of data set, namely a point cloud is often used. This point cloud contains data points in space and therefore already contains a certain definition of distance.

From this graph one can now use its k-cliques (complete subgraphs containing k nodes) to create a clique complex, where the clique complex of a graph G is equal to the Vietoris-Rips complex [Mun17] of the data set. Both the clique complex and the Vietoris-Rips complex are examples of simplicial complexes. To construct the clique complex of graph G, all cliques in this graph are marked and when all these possible cliques in the graph are found and marked, the high-dimensional holes can be identified. An example of such a clique complex with three 1-dimensional holes can be seen in Figure 2, where the nodes represent the cliques of size 1, the lines represent the 2-cliques and the yellow and green surfaces represent the 3- and 4-sized cliques, respectively.

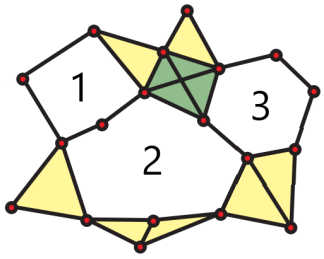


Figure 2: Clique complex example [GCD20]

2.1.2 Boundary map

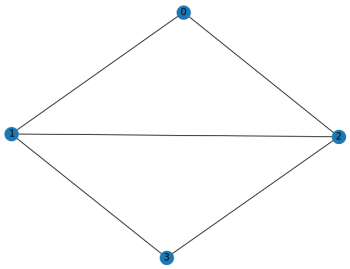
The boundary map (∂_k^G) is a linear operator of dimension $\binom{n}{k-1} \times \binom{n}{k}$, which tells us which cliques of size $k-1$ are part of a certain clique of size k. Here n represents the number of nodes in a graph G. As mentioned in Section 2.1.1, a clique is a complete subgraph where k defines the number of nodes in this subgraph. To create this boundary map, one checks for every clique of a chosen size k if this clique is part of the clique complex of G and what $(k-1)$-cliques are part of this k-clique. Now the boundary map is defined by the following expression:

$$\partial_k^G |j\rangle = \sum_{i=0}^{k} (-1)^i \widehat{|j(i)\rangle}. \tag{1}$$

This equantion is used to construct the boundary map corresponding to a graph G using a clique size k. In this equation the ket vector $|\cdot\rangle$ is used to represent bit strings which in turn represent cliques. This ket vector is part of the Dirac notation used in quantum computing and will be further explained in Section 2.2.1. In Equation 1, $|j\rangle$ contains the bit string of the integer value j which represents a different k-clique for each value of j and $\widehat{|j(i)\rangle}$ represents the bit string of the k-clique $|j\rangle$ where the i^{th} element of this bit string $|j\rangle$ is set to zero.

As mentioned before, a bit string representation is used in Equation 1. Each bit in these bit strings represents a node in the graph G, where an n-bit string solely filled with ones corresponds to a graph of size n. All cliques in this clique complex are also represented by n-bit strings, where bits that are set to 1 represent the nodes that are part of this clique and bits that are set to 0 represent the nodes that are not. Each k-clique of a complete graph of size n is thus represented by a n-bit string $|j\rangle$ with Hamming weight (the number of 1's in the bit string) k and each $(k-1)$-clique of this complete graph is represented by a n-bit string $\widehat{|j(i)\rangle}$ with Hamming weight $k-1$. Here all possible bit strings $|j\rangle$ of a complete graph of size n correspond to columns in the full boundary map and all possible bit strings $\widehat{|j(i)\rangle}$ of a complete graph of size n correspond to rows in this boundary map. In ascending order we now iterate trough these bit strings of Hamming weight k and if the k-clique corresponding to this bit string $|j\rangle$ is present in our clique complex of G, we can start checking which cliques of size $k-1$ are part of this k-clique. The elements in the boundary map for which $\widehat{|j(i)\rangle}$ exists in our clique complex are now filled in ascending order of i with alternating 1's and -1's. All columns corresponding to k-cliques which are not part of our clique complex, or elements whose $(k-1)$-clique is not part of a k-clique are set to zero.

Example
For a graph of size $n = 4$ and a chosen clique size $k = 3$ there are $\binom{4}{3} = 4$ possible k-cliques (or bit strings of Hamming weight k). In Figure 3a two of these four possible k-cliques for $k = 3$ are present in the clique complex of G, namely cliques $|0111\rangle$, corresponding to nodes '012' in the graph, and $|1110\rangle$, corresponding to nodes '123' in the graph. Now when we focus on clique $|0111\rangle$ and set the i^{th} 1 from its bit string in ascending order to zero, we get $(k-1)$-cliques $|0011\rangle$, $|0101\rangle$ and $|0110\rangle$. Because clique $|0111\rangle$ is the first possible k-clique in ascending order and this clique exists in our clique complex, the first column of the boundary map can be filled with 0's, 1's and -1's. Item (0, 0) of the boundary map is thus filled with a 1, representing k-clique $|0111\rangle$ and $(k-1)$-clique $|0011\rangle$. Entry (1, 0) is then filled with a -1, since 1 and -1 alternate, representing $|0111\rangle$ and $|0101\rangle$ and entry (2, 0) is 1 again, representing $|0111\rangle$ and $|0110\rangle$. The other $(k-1)$-cliques in the complete graph of size 4 are not part of this k-clique, thus its corresponding places in this column of the boundary map are set to zero. The k-clique $|1110\rangle$ is also part of the clique complex of G, containing $(k-1)$-cliques $|0110\rangle$, $|1010\rangle$ and $|1100\rangle$ and is filled in the same way as done for clique $|0111\rangle$. In this particular graph, $|0110\rangle$ is part of clique $|0111\rangle$ as well as clique $|1110\rangle$ and thus its row is nonzero in both columns corresponding to these k-cliques. As can be seen, k-cliques $|1011\rangle$ and $|1101\rangle$ are missing in the clique complex and so their columns are filled with zeros. This also goes for $(k-1)$-clique $|1001\rangle$, which row is thus filled with only zeros. Figure 3b shows the resulting boundary map.



(a) Example graph G of size $n = 4$

$$\partial_3^G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

(b) Resulting boundary map ∂_3^G

Figure 3

2.1.3 Combinatorial Laplacian

Using the boundary map as defined in Section 2.1.2, we now want to focus on the connectivity between cliques of a graph G. By Hodge theory [Fri98] we can use this boundary map and its transpose ($A^\intercal$ for which $[A^\intercal]_{ij} = [A]_{ji}$) to construct the combinatorial Laplacian, which is defined as:

$$\Delta_k^G = (\partial_k^G)^\intercal \partial_k^G + \partial_{k+1}^G (\partial_{k+1}^G)^\intercal, \tag{2}$$

giving us a linear operator of dimension $\binom{n}{k} \times \binom{n}{k}$. Hodge theory now also provides us with a way to find the k^{th} Betti number β_k^G (or number of k-dimensional holes), namely by taking the dimension of the kernel of this combinatorial Laplacian:

$$\dim \ker (\Delta_k^G) = \beta_k^G. \tag{3}$$

Calculating this k^{th} Betti number using equation 3 is the main quantity that the quantum topological data analysis algorithm wishes to compute.

2.1.4 Dirac operator

Another way of calculating the Betti number is by using the Dirac operator (B^G). This is a block matrix of the form:

$$B^G = \begin{pmatrix} 0 & \partial_1^G & 0 & .. & 0 \\ (\partial_1^G)^\intercal & 0 & \partial_2^G & 0 & : \\ 0 & (\partial_2^G)^\intercal & 0 & \ddots & 0 \\ : & 0 & \ddots & \ddots & \partial_{n-1}^G \\ 0 & .. & 0 & (\partial_{n-1}^G)^\intercal & 0 \end{pmatrix}.$$

This Dirac operator is a matrix of dimension $2^n - 1 \times 2^n - 1$ and can be used to calculate the k^{th} Betti number just like the previously mentioned combinatorial Laplacian since the square of the Dirac operator can be defined as:

$$B_G^2 = \begin{pmatrix} \Delta_0^G & 0 & .. & 0 \\ 0 & \Delta_1^G & 0 & : \\ : & 0 & \ddots & 0 \\ 0 & .. & 0 & \Delta_{n-1}^G \end{pmatrix}.$$

One can use projection operators to project all block matrices Δ_d^G in B_G^2 for which $d \neq k$ to zero and so create a matrix containing only Δ_k^G and zeros in the rest of the matrix. Thus using the Dirac operator is a more universal way of calculating Betti numbers where one Dirac operator can be used to calculate all k^{th} Betti numbers for which $k < n$ of a graph of size n. Unfortunately using the Dirac operator for quantum topological data analysis makes use of more qubits than when using the combinatorial Laplacian since the Dirac operator is larger than this combinatorial Laplacian. For this reason we chose to not use this Dirac operator in the presented research, which is focused on the use of quantum topological data analysis on near-term quantum computers, because these devices do not have many qubits available (see Section 2.2.9).

2.2 Quantum computing

This section tells us about the inner workings of a quantum computer and how it can modify the state of the memory of this quantum computer to perform calculations. We will also go over the limitations of current quantum computers to create a better understanding as to why the presented research is important.

2.2.1 Hilbert spaces and the Dirac notation

The mathematical formulation of quantum mechanics is somewhat different from the mathematical formulation used in classical mechanics. For quantum mechanics, and thus also quantum computing, Hilbert spaces and the Dirac notation are commonly used. Hilbert spaces ($\mathcal{H}$) specify the real or complex vector space used in quantum mechanical computations, consisting of any finite of infinite number of dimensions. Using the Dirac notation a classical vector v is written as $|v\rangle$. This is called a ket vector. The Hermitian conjugate $v^\dagger$ of this classical vector is represented in the Dirac notation using a bra vector $\langle v|$.

2.2.2 Qubits

Qubits (or quantum bits) are the building blocks of quantum computers. The states of these qubits specify the state of the quantum computer and by altering the state of the qubits, we alter the state of the quantum computer. A qubit can be in the state up, $|0\rangle$ and the state down, $|1\rangle$, and anywhere in between (represented by $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ as will be explained later in this paragraph). A qubit that is in between states these two states is in a superposition and only when a measurement is done will the qubit collapse to the up or down state.

The state of a single qubit can be visualized using the Bloch sphere representation as seen in Figure 4. Here the probability of the qubit being in the up or down state is represented by the angle θ. Thus when the qubit is at an angle of $\theta = 0$ for instance, it has 100 percent probability of being in the up state and when a qubit is at an angle of $\theta = \pi/2$ it has equal probability of being in the state $|0\rangle$ and $|1\rangle$. The (relative) phase of the qubit is represented by the angle ϕ, which is relevant when rotating the qubit.

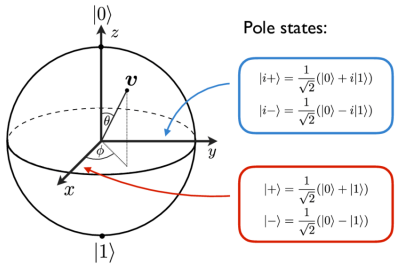


Figure 4: Bloch sphere [Ket16]

The state of a single qubit is a unit vector in the two-dimensional Hilbert space with complex amplitudes and is of the form:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}, \quad \text{where } \alpha, \beta \in \mathbb{C}.$$

The amplitudes α and β also specify the probability of the qubit collapsing to the up or down state, respectively. Thus the up state of a qubit can be represented by $\begin{pmatrix}1\\0\end{pmatrix}$ with a 100 percent probability of being in the up state and the state down is represented by $\begin{pmatrix}0\\1\end{pmatrix}$ with a 100 percent probability of being in the down state. The state of a qubit in between this up and down state is represented as a unit vector $\begin{pmatrix}\alpha\\ \beta\end{pmatrix}$ where α and β are both nonzero values.

Multi-qubit systems

Using this knowledge of a one-qubit system we can now create multi-qubit systems where qubits can interact with one another. A n-qubit system can be represented as a normalized, 2^n-dimensional vector and their joint state is given by the tensor product ($\otimes$) of the involved qubit states as

depicted below for a 2-qubit system:

$$|\psi\rangle \otimes |\chi\rangle = |\psi\chi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \otimes \begin{pmatrix} \gamma \\ \eta \end{pmatrix} = \begin{pmatrix} \alpha\gamma \\ \alpha\eta \\ \beta\gamma \\ \beta\eta \end{pmatrix}, \quad \text{where } \alpha, \beta, \gamma, \eta \in \mathbb{C}.$$

However, not all multi-qubit systems can be decomposed into a tensor product of local states again. If this is the case, the qubits of this system are entangled. We can for instance take a look at a 2-qubit system in the state $|\Phi^+\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, which is one of the four maximally entangled states called the Bell states [You20]. Now if the state of one qubit is known we instantly know the state of the other qubit, namely if the first qubit is in the state up in this example, the second qubit is as well and if the first qubit is in the state down, so is the second qubit.

2.2.3 Quantum gates

Quantum gates, or quantum logic gates, are the quantum equivalent of classical logic gates. On a classical computer logic gates specify how the state of the computer changes and so quantum gates specify how the state of a quantum computer changes. These quantum gates are unitary operators (U for which $UU^\dagger = U^\dagger U = I$) acting on quantum registers containing the states of the qubits in the system. But one important point where quantum computing differs from classic computing, is the way a state change is invoked. If a classical bit changes state, for instance from 0 to 1, the bit is simply overwritten and the information the bit previously held is lost. Quantum computing however allows the possibility of undoing certain calculations, where one can obtain the state held by a qubit before the application of a quantum gate. This can be done by applying the inverse of the quantum gate previously applied to a qubit, to this specific qubit.

If a quantum gate acts on one single qubit this gate is of dimension 2×2, as could be expected since now one can calculate the resulting qubit by multiplying the two dimensional vector of the input state of the qubit with the matrix of the quantum gate, resulting in a new two dimensional vector. This equation with quantum gate U has the form:

$$U\,|\psi_{in}\rangle = |\psi_{out}\rangle$$

A quantum gate can also act on multiple qubits and its matrix scales accordingly with respect to the tensor product structure of the state space: For a quantum gate acting on n input states, its matrix is of dimension $2^n \times 2^n$. If qubits are entangled for instance, is not be possible to separate their qubit states (as seen in Section 2.2.2). In this case it is not possible to apply quantum gates to a subsystem of these entangled qubits and thus the gate should be altered to be able to act on all qubit states which are entangled. Therefore one can take the tensor product of the identity gate combined with the quantum gate that one wants to use. The order of these gates is then decided by the order of the input qubits. So if one for instance wants to apply the X gate to a qubit '2' in the state $|\beta\rangle_2$, where the input state is $|\alpha\beta\gamma\rangle_{123}$, one should apply the $I \otimes X \otimes I$ gate (or IXI for short) to it. This way the IXI gate only acts like the X-gate on qubit '2' and acts like the identity on the other two qubits. This gate is now 1-local, because when a gate acts non-trivially on m qubits, this gate is called m-local.

It is possible that the program one wants to run on a quantum computer does not contain unitary matrices of the size $2^n \times 2^n$, as is often the case for the matrices used in the presented research. Then one needs to pad this unitary matrix to scale its size up to the smallest multiple of $2^n \times 2^n$ larger than the original matrix size to meet the requirements of the quantum computer.

Controlled gates

Controlled gates are a special type of quantum gates that are applied to two or more qubit states. Here we differentiate between the control qubit(s) and the target qubit(s). In case of the controlled-U gate, the target qubit(s) are the qubits on which the U gate will be applied. The control qubit(s) then specify if this gate is applied, namely this U-gate is only applied to the target qubits if the control qubits are in the state $|1\rangle$. The matrix representing a controlled-U gate applied to two qubits is of the form:

$$\text{Controlled-}U = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{pmatrix}, \quad \text{where } U = \begin{pmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{pmatrix}$$

Pauli gates

Four special gates worth mentioning are the Pauli gates (or matrices) and the identity matrix, as seen in Figure 5. Together, the sum of these four matrices with various real valued weights can represent all Hermitian matrices of size 2×2. When using a sum of tensor products of n of these Pauli matrices and the identity matrix, one can even form all Hermitian matrices of dimension $2^n \times 2^n$. How to form these Hermitian matrices of dimension $2^n \times 2^n$ using Pauli matrices and the identity matrix will be further explained in Section 2.2.6.

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Figure 5: Pauli gates

Converting between gates

Sometimes one needs a specific gate for a certain calculation (as we will see in Section 3.2). In this case quantum gate conversions can be used, which allow one to represent a unitary operator or quantum gate as a combination of different quantum gates. Two of these quantum gate conversions needed for the quantum topological data analysis algorithm are $X = HZH$ and $Y = SXS^\dagger$. Here H is the Hadamard gate, which can change the state of a qubit to and from a superposition state, and S is the phase gate which represents a rotation of $\phi = \pi/2$ on the Bloch sphere (see Section 2.2.2). Both the H-gate and S-gate are shown in Figure 6. $S^\dagger$ is the Hermitian conjugate of S.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

(a) Hadamard gate

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

(b) Phase gate

Figure 6: Examples of quantum gates

2.2.4 Hamiltonian simulation

The Hamiltonian H determines how a quantum system behaves in time, as stated in the Schrödinger equation $i\hbar\frac{d|\psi(t)\rangle}{dt} = H|\psi(t)\rangle$ [dW21]. Hamiltonian simulation makes use of this principle by simulating the evolution of a quantum state for a time t under a given Hamiltonian (which is a Hermitian operator). This can be accomplished by applying unitary operators, or specifically quantum gates, to one or multiple qubits, as seen in the following equation:

$$|\psi(t)\rangle = U|\psi(0)\rangle, \quad \text{where } U = e^{-iHt}$$

The Hamiltonian can be given to us in a few different ways including sparse access and local terms. When making use of sparse access the Dirac operator and projection operators are used to obtain the desired combinatorial Laplacian as explained in Section 3. Using local terms, the combinatorial Laplacian is directly calculated as done in Section 2.1.3 and one can implement Hamiltonian simulation using Trotterization which will be explained in Section 2.2.5.

2.2.5 Trotterization

When our goal is to implement $U = e^{iHt} = e^{i\sum_j H_j t}$ (for simplicity we will work with $U = e^{iHt}$ instead of $U = e^{-iHt}$), which can be hard to implement directly, Trotterization offers a solution. Trotterization, or the Lie-Suzuki-Trotter method [dW21], uses the fact that e^{A+B} is approximately equal to $e^A e^B$ for matrices A and B having a small norm. We can now decrease the norm of the Hamiltonian by dividing it by r (thus also decreasing the norm of the partial Hamiltonians), after which we can use this approximation to get a unitary operator of the form:

$$U = (e^{iHt/r})^r = (e^{iH_1 t/r + \ldots + iH_n t/r})^r = (e^{iH_1 t/r} \ldots e^{iH_n t/r} + E)^r.$$

This approximation uses an error E where the error term is a matrix of the same size as the unitary operator U, whose norm is $\|E\| = \mathcal{O}(t^2/r^2)$. The actual circuit that can now be implemented on a quantum computer is the approximation of this unitary operator U, namely:

$$\tilde{U} = (e^{iH_1 t/r} \ldots e^{iH_n t/r})^r.$$

So now the problem of Hamiltonian simulation is reduced to the problem of simulating partial Hamiltonians (see Section 3.1) which act non-trivially on only a few qubits. In the next sections we will see how one can use Pauli strings as explained in Section 2.2.6 to define the partial Hamiltonians and we will take a closer look at the implementation of this quantum circuit for $\tilde{U}$ (Section 3.2).

2.2.6 Pauli strings

As mentioned in Section 2.2.3, one can construct all Hermitian matrices of dimension $2^n \times 2^n$ by using a summation of tensor products of n Pauli matrices and the identity matrix with various real weights. This is done by the use of Pauli strings. Pauli strings are the tensor product of one or multiple matrices of the set $\{I, X, Y, Z\}$, where I is the identity matrix of size 2×2 and X, Y and Z are the Pauli matrices. The set $\mathcal{P}$ of all 4^n possible Pauli strings of dimension $2^n \times 2^n$ is defined by the following expression:

$$\mathcal{P} = \bigotimes_{i=0}^{n} \mathbf{p}^i, \quad \text{where } \mathbf{p}^i \in \{I, X, Y, Z\}.$$

All Hermitian matrices (in our case we will use a Hamiltonian H) of this size can now be created by a summation of Pauli strings in this set with real valued weights:

$$H = \sum_k \alpha_k P_k, \quad \text{where } P_k \in P,\ \alpha_k \in \mathbb{R}.$$

The weights of these Pauli strings vary and can be calculated using the Frobenious inner product (as seen in Equation 4). This Frobenious inner product can be calculated using the trace of the product of two matrices where A is the chosen matrix (of which we take the Hermitian conjugate) and B is the specific Pauli string:

$$\langle A, B \rangle_{\mathrm{F}} = \mathrm{Tr}\left(A^{\dagger} B\right). \tag{4}$$

2.2.7 Quantum circuits

A quantum circuit is a model for quantum computation where its corresponding circuit diagram visualizes the state of a quantum computer and the program executed by it. This program consists one or multiple quantum gates (Section 2.2.3) and n input qubits. In the quantum circuit in Figure 7 we can see four input qubits, in this case all in the state $|0\rangle$. Per qubit there is a horizontal line drawn, representing this qubit for the duration of this computation. All gates that are part of the program are then depicted on one or multiple of the lines originating from the input qubits. After the application of this program all n qubits are in the output state, which can be a superposition state. We can then measure the output qubits, collapsing the states of these qubits to either 0 or 1.

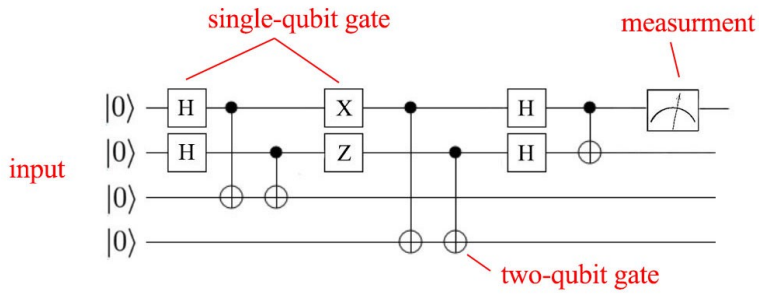


Figure 7: Quantum circuit example of four qubits [Hui18]

Measurement

To interpret the results of a quantum computation on a classical computer, one needs to measure the quantum system. This measurement collapses the measured qubits to classical bits after which these classical bits can be used on a regular classical computer. The likelihood of these qubits collapsing to 0 or 1 can be calculated using Born's rule [dW21]. Born's rule states that the probability of the qubit state collapsing to 0 or 1 is defined by the squared norm of the amplitudes α and β of $|0\rangle$ and $|1\rangle$, namely $\|\alpha\|^2$ and $\|\beta\|^2$, respectively.

2.2.8 Quantum phase estimation

Quantum phase estimation [NC00] is an algorithm with which one can estimate the phase ϕ of an eigenvalue (which can be written in the form $e^{2\pi i\phi}$) of a unitary matrix to a desired precision. This

algorithm takes as input t qubits in the state $|0\rangle$ together making up the first quantum register, a unitary operator U and m qubits defining an eigenstate $|\psi\rangle$ of this unitary operator U in the second quantum register. The output is then the estimation $|\tilde{\phi}\rangle$ of the phase of the eigenvalue of U, corresponding to eigenstate $|\psi\rangle$. When one wants to calculate how many eigenvalues are (approximately) equal to zero, one can scale this unitary matrix down by $1/\lambda_{max}$ (where λ_{max} is the maximal eigenvalue of U) to avoid multiples of 2π while keeping in mind that the algorithm has to be precise enough to correctly group a calculated phase as a zero or nonzero value. This algorithm has a total runtime of $\mathcal{O}(1/\epsilon^2)$ where ϵ is the additive precision. The circuit representation of the quantum phase estimation algorithm is shown in Figure 8.

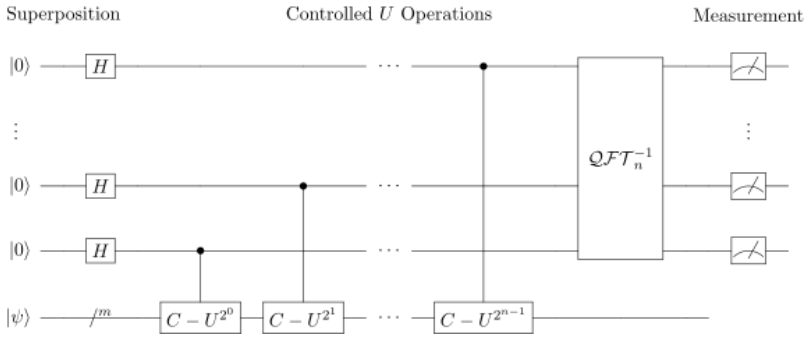


Figure 8: Quantum phase estimation circuit [qpe]

This algorithm consists of two main parts, namely performing the controlled-U operations and the application of the inverse quantum Fourier transform. For the first part of the quantum phase estimation procedure one needs two registers, the first one containing qubits in the up state and the second one containing as many qubits as necessary to store an eigenvector $|\psi\rangle$ of an input unitary operator U. The number of qubits in the first register is chosen based on the desired precision of the result. Each extra qubit in this register accounts for an extra digit precision in the resulting approximation of the phase. With the use of a Hadamard gate (Section 2.2.3) which, when applied to a qubit in the state $|0\rangle$, maps the qubit onto the superposition state $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$, we create an equal probability for the input qubits to be in the up or down state. Then we can apply the controlled-U gate as seen in Section 2.2.3 on target qubits in the second register and on a control qubit from the first register, where U is the operator whose phase we want to know. This leads to the eigenvalue of U being reflected in the phase of the control qubit. This phenomenon is called phase kickback. Per control qubit to which the controlled-U gate is applied, this gate is raised to successive powers of two, resulting in the following state of the first register:

$$\frac{1}{2^{t/2}}(|0\rangle + e^{2\pi i 2^{t-1}\phi}|1\rangle) \otimes (|0\rangle + e^{2\pi i 2^{t-2}\phi}|1\rangle) \otimes ... \otimes (|0\rangle + e^{2\pi i 2^{0}\phi}|1\rangle) = \frac{1}{2^{t/2}} \sum_{j=0}^{2^t-1} e^{2\pi i \phi j}|j\rangle .$$

Then the inverse quantum Fourier transform is applied to the first register. This inverse quantum Fourier transform has a runtime of $\mathcal{O}(t^2)$ where t is the number of qubits in the first register and it executes the transformation:

$$\frac{1}{2^{t/2}} \sum_{j=0}^{2^t-1} e^{2\pi i \phi j}|j\rangle|\psi\rangle \rightarrow |\tilde{\phi}\rangle|\psi\rangle ,$$

where $|\tilde{\phi}\rangle$ is the estimated phase. Now one can measure the first register to get the result $\tilde{\phi}$.

2.2.9 Limitations

Quantum computing technology is still in its infancy resulting in quite some restrictions when wanting to run quantum algorithms. For near-term computations fault-tolerant quantum computers are not yet available and therefore one uses noisy intermediate-scale quantum (NISQ) computers [BCLK+21], meaning it consists of a limited number of qubits which are not error-corrected and thus are subject to gate errors and quantum decoherence.

The bottleneck of quantum error correction [dW21] is the large number of qubits needed for this procedure. Unfortunately with only a limited number of qubits available, near-term quantum computers do not scale well enough to allow error correcting procedures. In 2019 a quantum computer using 53 qubits was revealed [AAB+19], which is still one of the largest quantum computers to this day. Two other factors limiting current quantum computers are gate errors and quantum decoherence, as mentioned before. Gate errors are caused by imperfect quantum gates which introduce errors in the output state of the qubits to which the gate is applied. This is still one of the biggest problems of quantum computers at this point in time. Quantum decoherence is the effect where the state of a quantum system, in our case qubits, is lost over time. This can happen anytime because of interactions of the qubit with the environment. Because of this, longer calculations can lead to greater errors. Therefore current quantum algorithms should be as insensitive as possible to gate errors and quantum decoherence.

2.3 Related work

This thesis is based on a paper on quantum topological data analysis [LGZ15] where the authors introduce the quantum approach to topological data analysis. Another related paper focuses on the complexity of the quantum topological data analysis algorithm and whether large speedups can be obtained when using quantum computers over classical computers for this algorithm [GCD20]. In this second paper the authors analyze what challenges exist for quantum topological data analysis on near-term devices, including the limited number of qubits available and its decoherence due to the use of too many quantum gates, and what possibilities of overcoming these obstacles there are.

3 Quantum topological data analysis

The original quantum topological data analysis algorithm [LGZ15] uses Hamiltonian simulation based on sparse access. This method uses oracles specified by quantum circuits, which allow one to query the values of its entries. However the downside of this sparse access method is the fact that it uses more than twice the qubits used in the Trotter-based method. Therefore, since NISQ computers are limited in terms of matrix size due to a maximum number of qubits, as explained in Sections 2.2.3 and 2.2.9, we will focus on Trotterization instead of sparse access. In this section the alterations needed to run the topological data analysis algorithm on a small-scale quantum computer will be explained, as well as the solutions to overcome these challenges.

3.1 Partial Hamiltonians

We will focus on simulating Hamiltonians using local terms. Therefore we want to split up these Hamiltonians as shown in Equation 5.

$$H = \sum_{j=1}^{m} H_j. \tag{5}$$

In this thesis I will refer to these elements H_j as partial Hamiltonians. Now each partial Hamiltonian acts non-trivially on only a few qubits and acts like the identity on the other qubits (see Section 2.2.3). When one of these partial Hamiltonians acts non-trivially on m qubits, this partial Hamiltonian is m-local.

3.2 Implementing topological data analysis on quantum computers

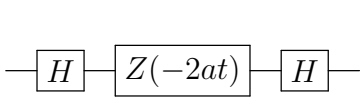
The question now arises of how Trotterization and Pauli decomposition all fit together on a quantum circuit. I will first explain how to represent the unitary operator $\tilde{U}$ on a quantum computer and I will then go into the details of executing the Trotter-based quantum topological data analysis algorithm.

To represent the unitary operator U on a quantum computer we start out with this operator U as declared in Section 2.2.4 and decompose its Hamiltonian (this will be explained in Section 3.3), after which one can use Trotterization to get the operator $\tilde{U}$. Now we can implement this $\tilde{U}$ on a quantum computer using the phase shift gate $Z(\theta)$ from Figure 9, since $U = e^{iaZt}$ is equal to $Z(-2at)$ with a being any constant (which in our case will represent the weight of a Pauli string). Using the quantum gate conversions as seen in Section 2.2.3 we have a way to implement the approximated unitary operator $\tilde{U}$, where the partial Hamiltonians of $\tilde{U}$ are represented by Pauli strings, on a quantum computer.

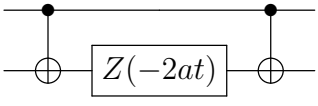
$$Z(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

Figure 9: Phase shift gate

In Figures 10a and 10b we see two examples of circuits, with the Hamiltonians being the Pauli strings X and ZZ which are 1-local and 2-local Hamiltonians (Section 3.1), respectively.



(a) $U = e^{iaXt}$ circuit



(b) $U = e^{iaZZt}$ circuit

Figure 10: Quantum circuit examples

Now we can go a step further as to choose a Hamiltonian consisting of multiple Pauli strings. One can then simulate $\tilde{U}$ by simply simulating each product of e individually and concatenating the resulting quantum circuits. An example of this can be seen in Figure 11.

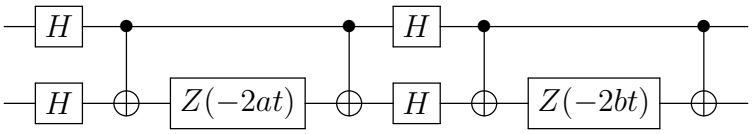


Figure 11: $U = e^{iaXXt}e^{ibZZt}$ circuit

To execute the Trotter-based quantum topological data analysis algorithm one uses quantum phase estimation. We start out with as many qubits in the first quantum register as needed to reach the precision required to differentiate a nonzero eigenvalue from a zero eigenvalue (or the low-lying spectrum as we will use, which is explained in Section 5.3). In the second register the eigenvector $|\psi\rangle$ starts out in the maximally mixed state $I/2^n$. To represent the controlled-$\tilde{U}$ operation, controlled versions of the tensor product of gates acting simultaneously on (a subsystem of) the qubit states in the second register are made as done in Section 2.2.3. Then these controlled gates are concatenated to represent this controlled-$\tilde{U}$ operation and afterwards the inverse quantum Fourier transform as explained in Section 2.2.8 is applied to the first register. To calculate the approximate Betti number one then estimates the normalized nullity $\beta_k^G/\dim\mathcal{H}_k^G$ up to an additive precision by running the quantum phase estimation algorithm m times, where the number of zero eigenvalues divided by m estimates this normalized nullity.

3.3 Pauli decomposition

Since we know how to decompose a chosen Hermitian matrix using Pauli decomposition, which decomposes this matrix into a summation of Pauli strings, this procedure can be applied to the combinatorial Laplacian. The combinatorial Laplacian is Hermitian and thus a valid matrix for Pauli decomposition. One can then use Trotter-based Hamiltonian simulation to implement the approximated unitary operator $\tilde{U}$ as defined in Section 2.2.5 on a quantum computer, where the combinatorial Laplacian represents the Hamiltonian. Since the combinatorial Laplacian is given to us in local terms we use Pauli decomposition on this combinatorial Laplacian so that each Pauli string (and its weight) represents a partial Hamiltonian.

Here we can also see why the partial Hamiltonians act on only a few qubits as mentioned in Section 2.2.4. Since the Pauli strings representing these partial Hamiltonians contain matrices from the set $\{I, X, Y, Z\}$ (Section 2.2.6), it is very likely that the identity matrix is part of this Pauli string. And as said before, when an identity gate is applied to a qubit, the qubit does not change states. Therefore we can say the partial Hamiltonian does not act on this qubit.

4 Research question

The research question naturally emerges when one takes a look at the size requirements of the topological data analysis algorithm while keeping in mind the capabilities and limitations (Section 2.2.9) of current quantum computers: *Can quantum topological data analysis be implemented by relying on space efficient Trotterization methods in Hamiltonian simulation?*

These experiments thus focus on the scaling of the number of nonzero Pauli strings in the graph size and clique size and the possibility of reducing the computational depth created by using Trotterization and Pauli decomposition. We decompose the combinatorial Laplacian using Pauli strings, where each Pauli string with a nonzero weight (or nonzero Pauli string) represents a partial Hamiltonian used to form the approximated unitary operator created by Trotterization. We investigate the computational depth created using Trotterization by taking a look at the number of nonzero Pauli strings in the decomposition and investigating the possibility of reducing this computational depth by removing Pauli strings while keeping the Betti number estimation within a chosen error margin. Also the possibility of reducing this error when estimating the Betti number is investigated. The results of these experiments can be found in Section 6.

5 Methods

The methods used for conducting the experiments of this thesis are described in this section. For the presented research we do not make use of actual quantum computers. Firstly this is the case because there are no quantum computers available for this research, but also because we investigate the number of nonzero Pauli strings and the error created by removing Pauli strings from the Pauli decomposition. These experiments must therefore be able to compare the results gained by using quantum procedures with exact results obtained by using classical computers.

5.1 Types of graphs

Although the topological data analysis algorithm is originally applied to data sets, we will not work with data sets in this thesis. Our focus will be on the optimization of the quantum topological data analysis algorithm and since converting data sets to graphs is a completely classical step in the process, it is not relevant for the presented research. Therefore generated graphs will be at the base of this research. Topological data analysis is not restricted to a certain types of graphs, thus the quantum topological data analysis algorithm should be compatible with all types of graphs.

In this thesis I focus on highly connected graphs. These are graphs which have a high percentage of its total possible edges and thus have a high chance of containing the desired cliques compared to graphs with a smaller percentage of edges present. For these graphs the quantum topological data analysis algorithm is the most efficient and these graphs give us a more representative model for graphs of larger sizes. Therefore I have chosen to work with graphs containing 90 percent of all its possible edges (rounded to its nearest integer value).

5.2 Algorithms

In this section the algorithms used for my experiments will be described. These algorithms are written in Python.

5.2.1 Generating graphs

For my experiments I have written code which generates random graphs using the Python library NetworkX [nx]. Here one starts with a complete graph of size n generated by NetworkX, after which we randomly select edges to be removed until the number of the edges that need to be removed to get a graph containing 90 percent of its edges is reached.

5.2.2 Boundary map algorithm

The algorithm that creates the boundary map takes as input a graph created by the methods in Section 5.2.1. It then finds all k- and $(k-1)$-sized cliques for a complete graph of size n and looks at how many of these cliques are in our graph, using functions from NetworkX. After this, the boundary map is filled using a similar technique as the one explained in Section 2.1.2. In my code however, I do not make use of bit strings to represent the cliques as done in the section on boundary maps, but work with an ordering defined by NetworkX. Therefore the boundary map is filled using the graph and subgraph representations of NetworkX instead of the bit string representation. However the boundary map created by the use of bit strings can be easily obtained by a simple permutation of rows and columns when one has the boundary map create by using NetworkX.

As mentioned above, the boundary map can also be created using the bit representation instead of using the functionalities of NetworkX. This code would fill the boundary map according to Equation 1. Using this bit string representation instead of the NetworkX representation reduces the computational complexity of this algorithm, resulting in a faster and more space efficient calculation. The reason why I did not use this technique is because the code of the boundary map algorithm had already been written using the NetworkX technique when I realized the bit string representation was also a viable option. Since it was not of great importance to use this bit string notation over the NetworkX technique, it was chosen to leave the boundary map code as is.

5.2.3 Combinatorial Laplacian algorithm

The combinatorial Laplacian (as seen in Equation 2) is easily implemented using a small for-loop which calculates ∂_k^G and ∂_{k+1}^G. However, the combinatorial Laplacian might need padding to fit on a quantum computer, as explained in Section 2.2.3. This is done by finding the smallest value of 2^n larger than the size of the combinatorial Laplacian. A matrix of size $2^n \times 2^n$ is then initialized with zeros and the combinatorial Laplacian is added to the left upper corner of this matrix, which gives us:

$$\text{Padded } \Delta_k^G = \begin{pmatrix} \Delta_k^G & 0 & .. & 0 \\ 0 & 0 & & : \\ : & & \ddots & : \\ 0 & .. & .. & 0 \end{pmatrix}$$

5.2.4 Pauli decomposition algorithm

Firstly we use the bit string representation to identify all possible combinations of Pauli strings which are of the same size as the (padded) combinatorial Laplacian. All possible bit strings of length 2^{2n} (for a combinatorial Laplacian of dimension $2^n \times 2^n$) are generated, where each of the four Pauli matrices is represented by a 2 bit string, namely $I =$ '00', $X =$ '01', $Y =$ '10' and $Z =$ '11'. This results in a tensor product of n Pauli matrices, after which the Frobenious inner product (as seen in Equation 4) can be used to calculate the weight of each Pauli string. In our case the real weight α_i of Pauli string A_i is calculated using $\alpha_i = \frac{1}{2^n} \cdot Tr(\Delta_k^G A_i)$, where $\Delta_k^G = (\Delta_k^G)^\dagger$ since Δ_k^G is Hermitian and $\frac{1}{2^n}$ is the normalization factor.

A certain percent of nonzero Pauli strings, chosen by the user, with the smallest weights are then marked. Afterwards the combinatorial Laplacian is reconstructed using only the unmarked Pauli strings (this way decreasing the computational depth), resulting in a perturbation of the original combinatorial Laplacian which hopefully still gives a correctly estimated Betti number as result despite its perturbation.

5.2.5 Quantum phase estimation algorithm

Using Cirq [cir], a Python software library for writing quantum circuits, I have worked on taking the quantum phase estimation algorithm (Section 2.2.8) up into my code. Later during my research however, the code appeared not relevant for this thesis. Thus, even though the quantum phase estimation algorithm is an important part of quantum topological data analysis, it was decided not to be used in the presented research.

5.3 Low-lying spectrum

In the presented research one of the experiments is to compare the nullity of the combinatorial Laplacian and the perturbed matrix. Here the nullity of the perturbed matrix is defined as the low-lying spectrum, or all eigenvalues of this matrix which are below the smallest eigenvalue λ_{min} of the combinatorial Laplacian.

5.4 Implementation of a prefactor

To reduce the error in Betti number estimation created by discarding Pauli strings we introduce the notion of a prefactor. Where we have previously chosen to set the threshold of the low-lying spectrum at λ_{min} (see Section 5.3), we will now set the threshold of the low-lying spectrum at $\tau\lambda_{min}$ where τ is a value between 0.05 and 1.0. Here $\tau\lambda_{min}$ with a prefactor of $\tau = 1.0$ is of course simply λ_{min}. This prefactor will thus result in a decrease of the threshold of the low-lying spectrum which could lead to a reducution of the error in the estimation of the Betti number.

5.5 Data analysis

Using the algorithms as mentioned in Section 5.2, I have investigated various aspects of the research question as described in Section 4. Using randomly generated graphs as a starting point, I have

visualized the acquired data using the Python package Matplotlib resulting in the plots seen in the next section.

6 Results

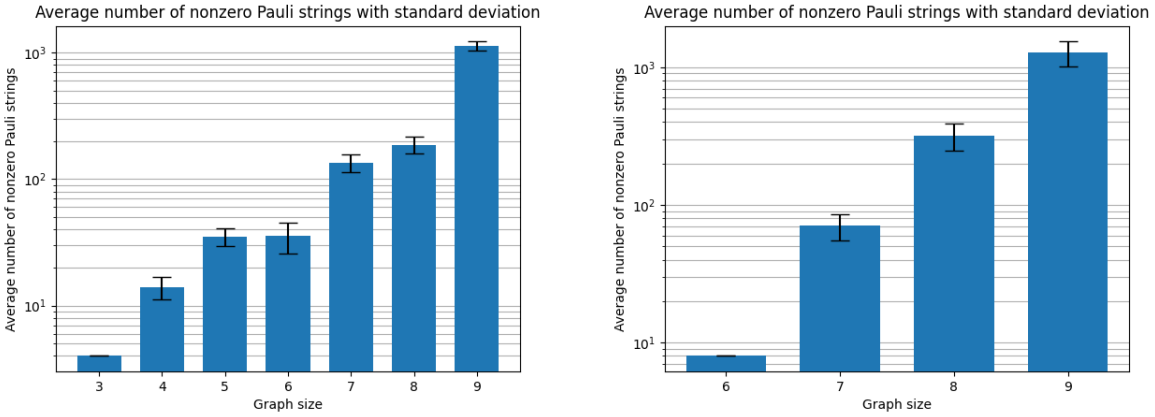
In this section the results of my research will be discussed. First we will look at the scaling of the number of Pauli strings with nonzero weight (or nonzero Pauli strings) used, given certain graph sizes and clique sizes. Then the error of the nullity approximation is researched and later we look into the possible use of a prefactor for the eigenvalue threshold. Due to the limited computing power of the classical computers used, it was not possible to work with graphs larger than $n = 9$, thus also limiting the clique size to at most $k = 8$. Per variable measured 100 instances of graphs are run to create a representative model. These 100 graphs were the same for each clique size k and for each percentage of Pauli strings removed. Only when changing the graph size n, 100 new graphs had to be generated. By using the same graphs when working with a changing clique size and a changing percentage of removed Pauli strings, we eliminate the possibility of the graph sampling being a variable in the results.

6.1 Number of nonzero Pauli strings

The number of nonzero Pauli strings for combinatorial Laplacians with a constant and variable clique sizes is investigated in this section. Here we will also take a look at how the computational depth of the topological data analysis algorithm differs between these clique sizes.

6.1.1 Constant clique size

By theory we know that the maximal number of Pauli strings used in the decomposition grows proportional to the size of the combinatorial Laplacian. The combinatorial Laplacian is of dimension $\binom{n}{k}$ rounded up to the nearest multiple of 2^i. Because $\binom{n}{k} \sim n^k$ given a constant value for k, the growth of this binomial coefficient and therefore the growth of the size of the combinatorial Laplacian is upper bounded by a polynomial. We know that for a linear operator of dimension 2^i there are $4^i = 2^{2i}$ possible Pauli strings, therefore when the size of combinatorial Laplacians grows polynomially in n (as it does with a constant value for the clique size k), the number of possible Pauli strings for a graph size n is also upper bounded by a polynomial.



(a) Average number of Pauli strings with nonzero weight for clique size $k = 2$

(b) Average number of Pauli strings with nonzero weight for clique size $k = 5$

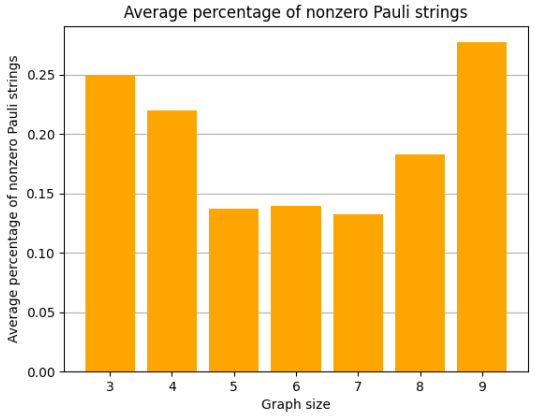
Figure 12: Average number of Pauli strings with nonzero weight with standard deviation

Figure 12 shows the number of nonzero Pauli strings used in the Pauli decomposition of combinatorial Laplacians for a variable graph size. Unfortunately graphs larger than $n = 9$ could not be run because of the limited computational power of the computer used to run the plots. Thus due to the limited number of data points in these plots a clear, polynomial scaling could not be shown. However it is expected that when increasing n, increasingly more graph sizes will be scaled up to the same multiple of 2^i which results in a polynomial scaling.

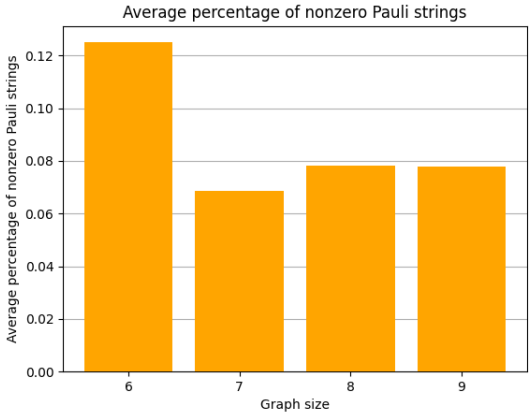
In Figure 12a we see that the number of nonzero Pauli strings for graph sizes $n = 5$ & 6 and $n = 7$ & 8 are very close together. For these graph sizes the (padded) combinatorial Laplacians are of the same size since the dimensions of these combinatorial Laplacians are rounded up to the same multiple of 2^i, namely 16×16 for $n = 5$ & 6 and 32×32 for $n = 7$ & 8. For these two groups of graph sizes the number of possible Pauli strings are also the same since the number of possible Pauli strings depends on the combinatorial Laplacian size as mentioned earlier. Still a small variance in the number of nonzero Pauli strings for these two groups of graph sizes can be seen in the figure above and more clearly in Figure 13a. This is most likely due to the sparseness of the combinatorial Laplacian which we will expain in Section 6.1.3.

Percentage of nonzero Pauli strings

When one compares the number of nonzero Pauli strings in Figure 12 with the maximal number of Pauli strings possible for the graph sizes used in this figure (which is 4^i as mentioned before), it is clear that in the plots of Figures 12 and 13 only a small percentage of Pauli strings have nonzero weight. This is fortunate because only the nonzero Pauli strings contribute to the computational depth of the algorithm and therefore less nonzero Pauli strings being used in the decomposition result in a smaller computational depth.



(a) Average percentage of Pauli strings with nonzero weight for clique size $k = 2$



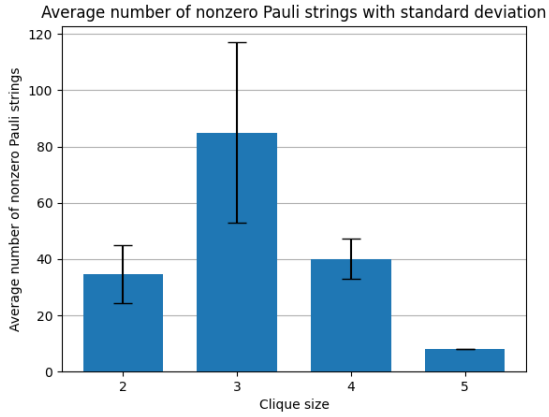
(b) Average percentage of Pauli strings with nonzero weight for clique size $k = 5$

Figure 13: Average percentage of nonzero Pauli strings

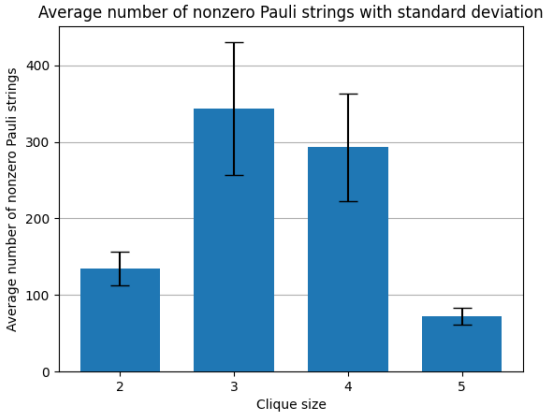
When taking a closer look at the percentage of Pauli strings which are nonzero for a given combinatorial Laplacian as done in Figure 13, it it hard to spot a trend in the percentage of nonzero Pauli strings for a constant value of k. However, when comparing Figures 13a and 13b we can see that the percentage of nonzero Pauli strings used in the decomposition of the combinatorial Laplacian decreased when using $k = 5$ compared to when using $k = 2$.

6.1.2 Variable clique size

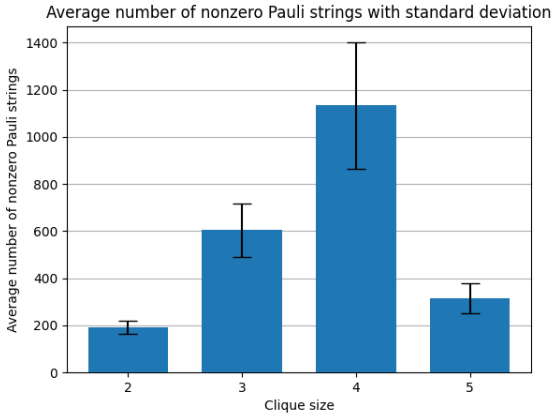
Now that we know the scaling of the number of Pauli strings for a constant value for k we investigate the number of Pauli strings for a variable clique size. For clique sizes of $k = n/2$ rounded up or down the number of possible Pauli strings is at a maximum, since for this clique size the size of the combinatorial Laplacian is the largest. Namely for this value of k the combinatorial Laplacian is of dimension $\binom{n}{\lfloor \frac{n}{2} \rfloor} \times \binom{n}{\lfloor \frac{n}{2} \rfloor} = \binom{n}{\lceil \frac{n}{2} \rceil} \times \binom{n}{\lceil \frac{n}{2} \rceil}$ scaled up to the smallest multiple of 2^i larger than these binomial coefficients. For any combinatorial Laplacian this binomial coefficient is upper bounded by n^k and where this n^k scales polynomial in n for a constant value for k, it scales exponentially for any variable k depending on n. Thus for a variable clique size the binomial coefficient is upper bounded by $\binom{n}{\frac{n}{2}} \sim n^{n/2}$. Using Stirling's approximation [Ege14] however, we know $\binom{n}{\frac{n}{2}} \sim \frac{2^{n+1}}{\sqrt{2\pi n}}$ and thus we can reduce this upper bound from $n^{n/2}$ to $\frac{2^{n+1}}{\sqrt{2\pi n}}$, resulting in a more fitted curve. Since the (padded) combinatorial Laplacian size scales with the binomial coefficient and the possible number of Pauli strings scales like 2^{2i} in respect to a combinatorial Laplacian of size $2^i \times 2^i$, we can state that the number of possible Pauli strings also scales exponentially in n, proportional to $(\frac{2^{n+1}}{\sqrt{2\pi n}})^2$ for a variable clique size. Since $k = n/2$ gives us the largest possible combinatorial Laplacian size and therefore the largest number of possible Pauli strings for the decomposition, a scaling with $\frac{2^{n+1}}{\sqrt{2\pi n}}^2$ is the worst possible scaling in n for the number of possible Pauli strings.



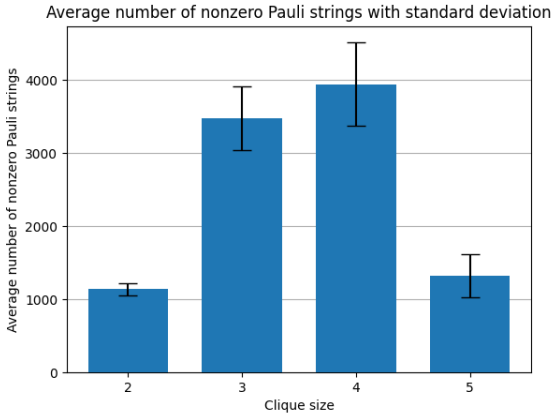
(a) Average number of Pauli strings with nonzero weight for graph size $n = 6$



(b) Average number of Pauli strings with nonzero weight for graph size $n = 7$



(c) Average number of Pauli strings with nonzero weight for graph size $n = 8$



(d) Average number of Pauli strings with nonzero weight graph size $n = 9$

Figure 14: Average number of Pauli strings with nonzero weight with standard deviation

In Figure 14 the number of nonzero Pauli strings for a variable clique size k is shown where one can see a maximum in the number of nonzero Pauli strings given a graph size n and a clique size $k = \lfloor \frac{n}{2} \rfloor$. This is to be expected since for this value of k the combinatorial Laplacian is the largest as mentioned before and thus the number of possible Pauli strings is also at a maximum. However, when n is uneven the value for $k = n/2$ is not an integer. This k will now have to be rounded up or down to get a combinatorial Laplacian of dimension $\binom{n}{int(\frac{n}{2})} \times \binom{n}{int(\frac{n}{2})}$. Interestingly, Figure 14 shows a larger number of nonzero Pauli strings for $k = \lfloor \frac{n}{2} \rfloor$ compared to for $k = \lceil \frac{n}{2} \rceil$. This phenomenon will be discussed in Section 6.1.3.

As mentioned above, the scaling of the number of possible Pauli strings in n is exponential when using a variable value for clique size k. In Figure 15 we take a look at the number of nonzero Pauli strings used in the decomposition of the combinatorial Laplacian for clique size $k = \lfloor \frac{n}{2} \rfloor$. In this figure a clear exponential trend for the number of nonzero Pauli strings in n can be seen.

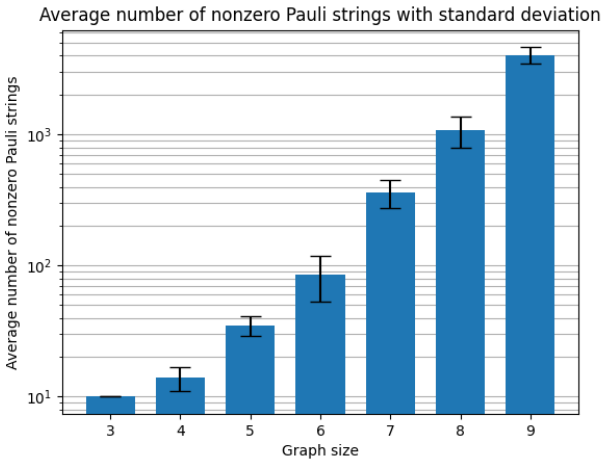


Figure 15: Average number of Pauli strings with nonzero weight per graph size n for clique size $k = n/2$

Percentage of nonzero Pauli strings

The percentage of nonzero Pauli strings for this clique size of $k = \lfloor \frac{n}{2} \rfloor$ per graph size is shown in Figure 16. Here we see that the percentage of nonzero Pauli strings has an overall downward trend, which could mean that the number of nonzero Pauli strings does not scale linearly, but inverse exponentially with $(\frac{2^{n+1}}{\sqrt{2\pi n}})^2$. However the number of nonzero Pauli strings for graph size $n = 9$ is an outlier, which could mean that the overall trend of this plot is not solely downward. Because of the maximum graph size of $n = 9$, we can unfortunately not confirm or reject this trend.

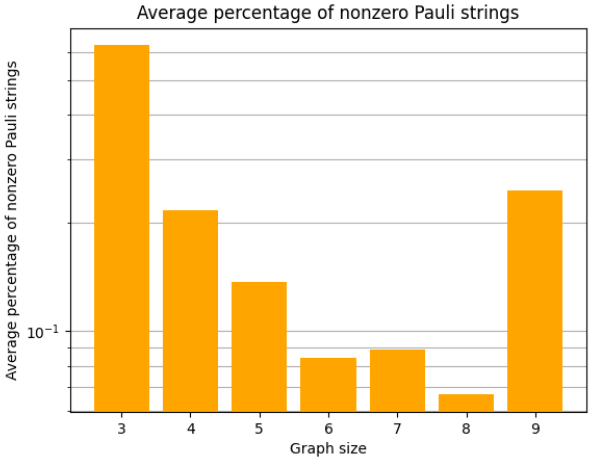


Figure 16: Average percentage of Pauli strings with nonzero weight per graph size n for clique size $k = n/2$

6.1.3 Relevance of padding to the number of nonzero Pauli strings

The combinatorial Laplacian used in these calculations are sparse, meaning that most elements in this matrix are zero. This is because of two reasons. Firstly, the boundary maps ∂_k^G and ∂_{k+1}^G of

which the combinatorial Laplacian is composed are heavily padded with zeros. Namely all rows and columns in this boundary map corresponding to k- and $(k-1)$-sized cliques respectively, which are not present in a given clique complex are set to zero (see Section 2.1.2). This we will call inner padding. Secondly, the combinatorial Laplacian is also padded up to the smallest multiple of 2^i larger than the unpadded combinatorial Laplacian size (Section 5.2.3), which we will call outer padding. In the next paragraph is becomes clear that the amount of inner padding is a relevant factor for the number of nonzero Pauli strings used in the decomposition of this combinatorial Laplacian. It would therefore be only logical that the amount of outer padding also affects the number of nonzero Pauli strings, but no clear evidence of this relevance of outer padding was found. However, an exact quantification of the relevance of both this inner and outer padding is not known and could be subject to further research.

To see the relevance of the amount of inner padding on the number of nonzero Pauli strings we take a look at Figure 14b. For graph size $n = 7$ and clique sizes $k = 3$ and $k = 4$ in this figure, the size of the unpadded combinatorial Laplacians are identical since $\binom{7}{3} = \binom{7}{4} = 35$ and thus these combinatorial Laplacians are rounded up to the same multiple of 2^i. Because of this same size combinatorial Laplacian, the maximal number of Pauli strings is also the same for both these clique sizes. However, in this figure one can see a larger number of nonzero Pauli strings for $k = 3$ cliques than for $k = 4$ cliques. Since a clique is a complete subgraph (a clique of size 4 always contains 4 cliques of size 3) we know that there are more cliques of size 3 present in this clique complex than 4-cliques and therefore there is less inner padding present in the combinatorial Laplacian for clique size $k = 3$ than $k = 4$. This pattern also holds for the number of nonzero Pauli strings of clique sizes $k = 3$ and $k = 5$ for a graph size $n = 8$ and for clique sizes $k = 4$ and $k = 5$ for a graph size $n = 9$ in Figure 14. Therefore one could reason that less inner padding equals the use of more nonzero Pauli strings in the decomposition.

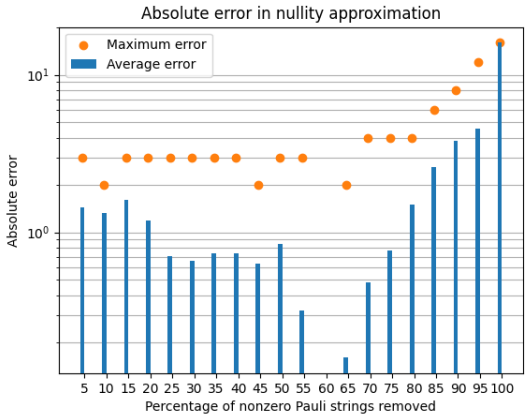
One would assume that if less inner padding equals more nonzero Pauli strings, less outer padding would also result in the use of more nonzero Pauli strings in a decomposition of the combinatorial Laplacian. For clique size $k = 2$ we know how many cliques are in the used graphs since 2-cliques are edges in these graphs and as mentioned in Section 5.1, the graphs used for the presented research contain 90 percent of its possible edges. Therefore the number of edges for a n-sized graph is $0.9 \cdot \binom{n}{2}$. However in the graphs for clique size 2 of the presented research, no two graph sizes have the same size combinatorial Laplacian and amount of inner padding. Therefore we can not isolate the amount of outer padding as a single variable and thus we can not say anything about the relevance of the amount of outer padding to the number of nonzero Pauli strings used.

6.2 Discarding Pauli strings with small weights

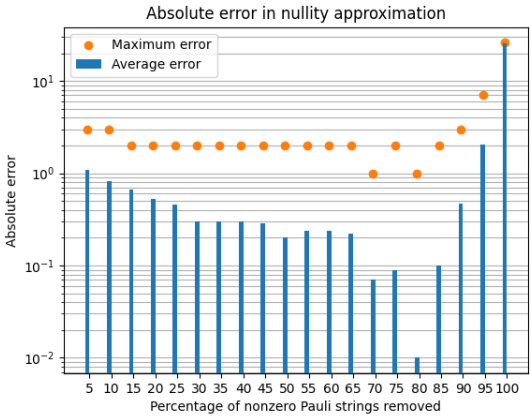
To decrease the computational depth created by the use of Trotterization, we look into the possibilities of dropping a certain percentage of nonzero Pauli strings used in the Pauli decomposition of the Hamiltonian (or combinatorial Laplacian in our case). As explained in Section 3.3, the combinatorial Laplacian can be decomposed into a summation of Pauli strings and their weights, where each Pauli string with nonzero weight accounts for a separate combination of quantum gates. These separate gate combinations can then be concatenated to create the full circuit representing the unitary operator $\tilde{U}$ (Section 3.2). Thus by removing a nonzero Pauli string, its corresponding

combination of quantum gates is removed from the full circuit, reducing the computational depth. Therefore, if we can remove a large percentage of nonzero Pauli strings while keeping the average error in the nullity approximation as low as possible, we can decrease the computational depth by an equally large percentage.

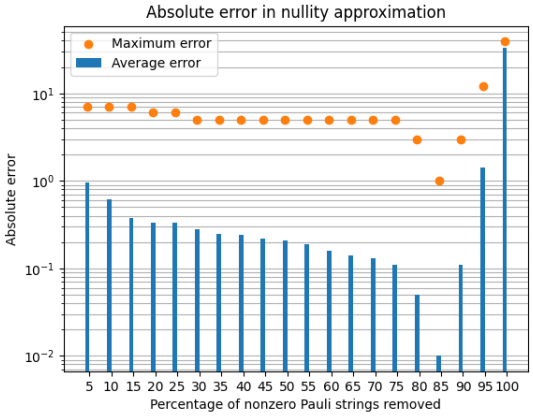
Since combinatorial Laplacians with a clique size $k = \lfloor \frac{n}{2} \rfloor$ and a given graph size n use the most nonzero Pauli strings as discussed in Section 6.1.2, these combinatorial Laplacians will benefit most from any decrease in the computational depth of the quantum topological data analysis algorithm. Also when k scales with n, the quantum topological data analysis can achieve super-polynomial speedups over classical methods [GCD20]. Therefore the focus of this experiment was on combinatorial Laplacians of dimension $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ scaled up to the smallest multiple of 2^i.



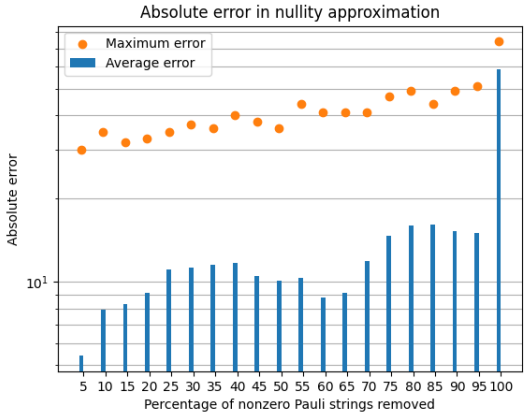
(a) Absolute error for graph size $n = 6$ and clique size $k = 3$



(b) Absolute error for graph size $n = 7$ and clique size $k = 3$



(c) Absolute error for graph size $n = 8$ and clique size $k = 4$



(d) Absolute error for graph size $n = 9$ and clique size $k = 4$

Figure 17: Absolute error in nullity approximation

Figure 17 shows the average absolute error over 100 graphs per percentage of Pauli strings removed and the maximum absolute error per percentage of Pauli strings removed for the nullity

approximation. This figure shows interesting results for the use of Pauli decomposition for near-term quantum topological data analysis. Namely one can remove up to around 70 percent of the nonzero Pauli strings while still keeping the maximal error and average error at a stable value, thus creating the opportunity to decrease the computational depth in exchange for a small error in the approximation. However, we see a slight decrease in the average error when removing a higher percentage of Pauli strings (up until a minimum in the average error at around 60 to 80 percent of Pauli strings removed, depending on the graph size). This is a peculiar feature and the reason for this decrease might be interesting to investigate in further research.

6.3 The use of a prefactor

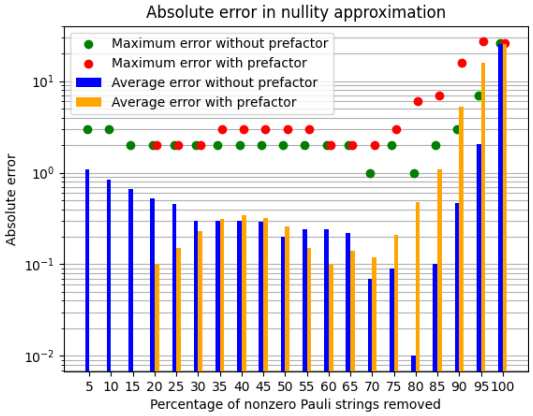
When removing a certain percentage of nonzero Pauli strings one is bound to get an error in the Betti number approximation as can be seen in the previous section. As can be seen in Figure 17, the maximal and average absolute error are not at a stable, low value for all graph sizes. We therefore look into the possibility of reducing this error as much as possible.

If we compare the eigenvalues of the perturbed combinatorial Laplacian, the matrix where we removed a certain percentage of its nonzero Pauli strings, with the eigenvalues of the original combinatorial Laplacian a small perturbation in the eigenvalues can be seen. Namely the zero eigenvalues of the original matrix often get scaled up to a small (positive or negative) nonzero value. This is not a problem since we measure the low-lying spectrum (see Section 5.3) of the perturbed matrix. However, we also see certain nonzero eigenvalues being scaled down towards zero because of the removal of a percentage of Pauli strings with nonzero weight. These scaled down, nonzero eigenvalues might now fall below the threshold of the low-lying spectrum, causing some of the errors one sees in Figure 17. Therefore one can modify the threshold of the low-lying spectrum by multiplying λ_{min} by a prefactor as seen in Section 5.4, essentially shifting the smallest nonzero eigenvalue of this combinatorial Laplacian down in order to keep the nonzero eigenvalues of this perturbed matrix above the threshold.

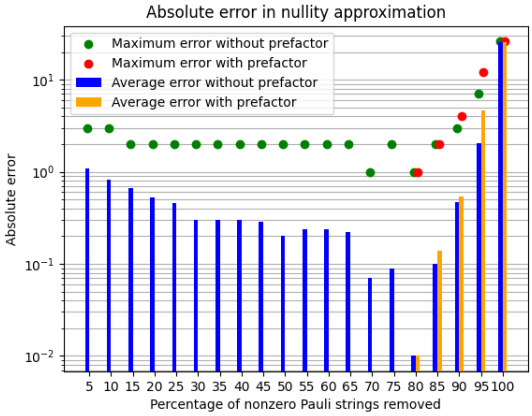
In this section we take a look at the optimal value for the prefactor. Then we plot the absolute error in nullity approximation for graph size $n = 6$ to $n = 9$ with corresponding clique size $k = \lfloor \frac{n}{2} \rfloor$, making use of this optimal prefactor to investigate whether this prefactor changes the error of the Betti number approximation.

6.3.1 The optimal prefactor

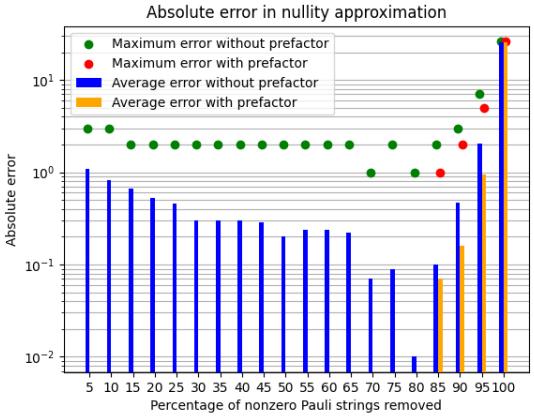
In figure 18 the absolute error in nullity approximation with and without a prefactor is compared for a graph size of $n = 7$ and clique size $k = 3$. Here the absolute error in nullity approximation with the use of a prefactor of 0.25, 0.50 and 0.75 is compared to the absolute error in nullity approximation without the use of a prefactor. From this figure and Appendix A which shows the absolute error in nullity approximation with a prefactor of 0.05 up to 0.95 compared to the absolute error in nullity approximation without prefactor, we can see that a prefactor of around 0.75 is optimal.



(a) Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.25



(b) Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.50



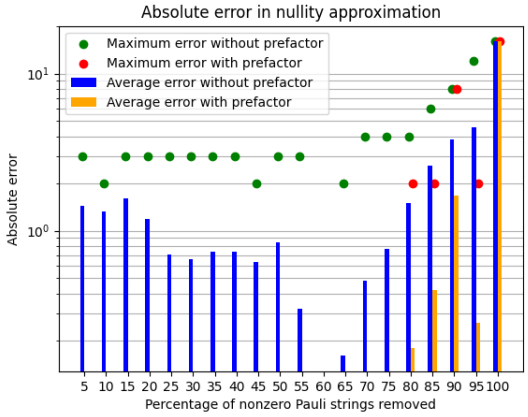
(c) Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.75

Figure 18: Absolute error in nullity approximation with a prefactor

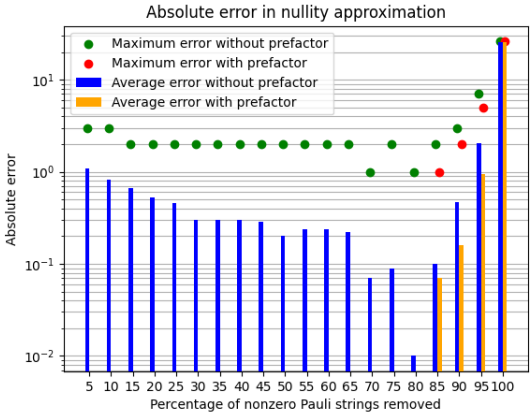
6.3.2 Error in nullity approximation when using a prefactor

Now we can compare the error in nullity approximation using a prefactor of 0.75 with the error in nullity approximation without the use of a prefactor for different graph sizes. In Figure 19 the result of this comparison is shown for graph sizes $n = 6$ to $n = 9$ with a clique size $k = \lfloor \frac{n}{2} \rfloor$. In this figure we can see that not only for $n = 7$ the prefactor of 0.75 gives an amazing decrease of the maximal and average absolute error, but for other graph sizes this prefactor greatly reduces the absolute error as well. For graph size of $n = 9$ however, the average error increases because of the decrease of the threshold of the low-lying spectrum (see Section 5.3). This is due to the fact that for graph size $n = 9$, the smallest nonzero eigenvalue has a significantly smaller absolute value than the smallest nonzero eigenvalue of the other graph sizes in Figure 19. Because of this smaller absolute value of the smallest nonzero eigenvalue of graph size $n = 9$, many zero eigenvalues get scaled up to above this threshold of λ_{min} resulting in the large error as seen in Figure 19d. When one now decreases the threshold of λ_{min} to $0.75 \cdot \lambda_{min}$, the nonzero eigenvalues which get scaled

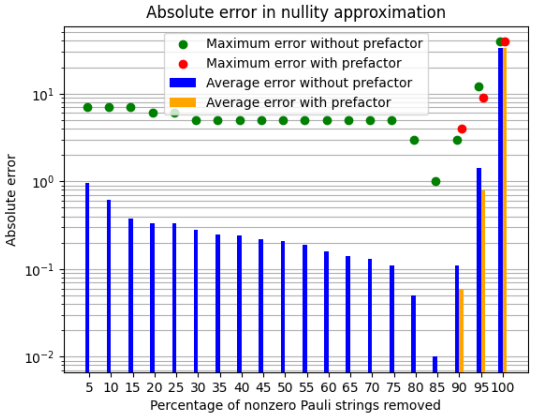
down do not get counted as a zero anymore, but many more zero eigenvalues get counted as a nonzero eigenvalue because of this decrease of the threshold, therefore ultimately increasing the average error.



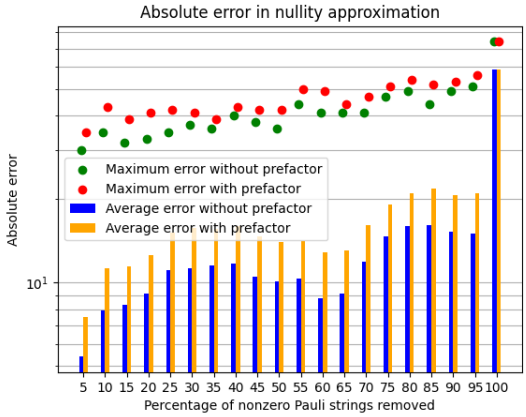
(a) Absolute error for graph size $n = 6$ and clique size $k = 3$ with a prefactor of 0.75



(b) Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.75



(c) Absolute error for graph size $n = 8$ and clique size $k = 4$ with a prefactor of 0.75



(d) Absolute error for graph size $n = 9$ and clique size $k = 4$ with a prefactor of 0.75

Figure 19: Absolute error in nullity approximation with prefactor

The average error percentage $\mathcal{E}_k^n$ over all percentages of Pauli strings removed (except for 100 percent or Pauli strings removed since this just gives the exact nullity of the combinatorial Laplacian) for graph size n and clique size k with and without the use of a prefactor is calculated using $\mathcal{E}_k^n = E_{k,avg}^n / \dim(\mathcal{H}_k^n)$, where $E_{k,avg}^n$ is the average absolute error. The average error percentages belonging to Figure 19 are shown in the following table:

n	$\mathcal{E}_{n/2}^n$ without prefactor (%)	$\mathcal{E}_{n/2}^n$ with prefactor $\tau = 0.75$ (%)
6	4.0	0.4
7	0.7	0.1
8	0.3	0.0
9	8.8	12.1

6.4 Discussion

One important restriction of this research is the limitation of the graph size n and clique size k. The combinatorial Laplacian increases vastly in size as mentioned in Section 6.1, therefore computations of graph size 10 combined with a clique size of 5 were already too computationally expensive, leading to a shutdown of the program. Namely, for $n = 10$ and $k = 5$ the combinatorial Laplacian is of dimension 256×256, whereas the largest combinatorial Laplacian that has been calculated for the presented research is of dimension 128×128 for $n = 9$ and $k = 5$. This resulted in only a limited amount of data being collected, making data analysis for this thesis more difficult.

7 Conclusion and further research

Trotterization and Pauli decomposition offer a reliable and space efficient ways of approximating the k^{th} Betti number for quantum topological data analysis without an overwhelming computational depth. With the use of classical pre-processing methods and discarding up to 70 percent of the Pauli strings with nonzero weight for various graph and clique sizes one is able to greatly decrease the computational depth of the quantum topological data analysis algorithm. A prefactor of 0.75 then greatly reduces the error percentage of the average absolute error created by discarding nonzero Pauli strings for certain graphs. Therefore Hamiltonian simulation via Trotterization gives a reliable near-term solution for implementing quantum topological data analysis on NISQ computers.

The padding of the combinatorial Laplacian seems to be an important factor in the scaling of the number of nonzero Pauli strings in the graph size n. Therefore looking further into this padding could give us a more clear view of the scaling of this umber of nonzero Pauli strings in n and thus what sizes of data sets can be used for topological data analysis using NISQ computers. Also the error in Betti number approximation can be an interesting topic for further research. We have seen that discarding nonzero Pauli strings from the Pauli decomposition can lead to an unwanted increase in this error and thus investigating for which graph sizes these errors are large (or for which graph sizes the smallest nonzero eigenvalues are small) can tell us when we can reliably remove a large percentage of nonzero Pauli strings.

References

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando Brandao, David Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, and John Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 10 2019.

[BCLK+21] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum (nisq) algorithms, 2021.

[cir] Cirq. https://quantumai.google/cirq.

[dW21] Ronald de Wolf. Quantum computing: Lecture notes, 2021.

[Ege14] Steffen Eger. Stirling's approximation for central extended binomial coefficients. *The American Mathematical Monthly*, 121(4):344, 2014.

[Fri98] J. Friedman. Computing betti numbers via combinatorial laplacians. *Algorithmica*, 21:331–346, 1998.

[GCD20] Casper Gyurik, Chris Cade, and Vedran Dunjko. Towards quantum advantage via topological data analysis, 2020.

[Hui18] Jonathan Hui. Qc — programming with quantum gates (single qubits), 2018.

[Ket16] Andreas Ketterer. *Modular variables in quantum information*. PhD thesis, 10 2016.

[LGZ15] Seth Lloyd, Silvano Garnerone, and Paolo Zanardi. Quantum algorithms for topological and geometric analysis of big data, 2015.

[Mun17] E. Munch. A user's guide to topological data analysis. *Journal of learning Analytics*, 4:47–61, 2017.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[nx] Networkx: Network analysis in python. https://networkx.org/.

[qpe] Quantum phase estimation circuit. https://en.wikipedia.org/wiki/Quantum_phase_estimation_algorithm#/media/File:PhaseCircuit-crop.svg.

[You20] Peter Young. Generating and measuring bell states, 2020.

A Optimal prefactor for nullity approximation

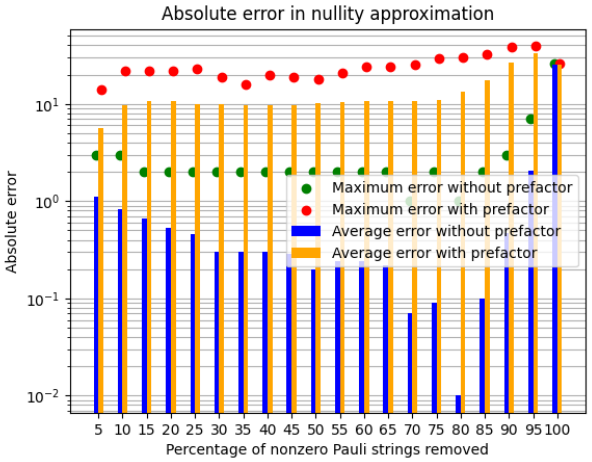


Figure 20: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.05

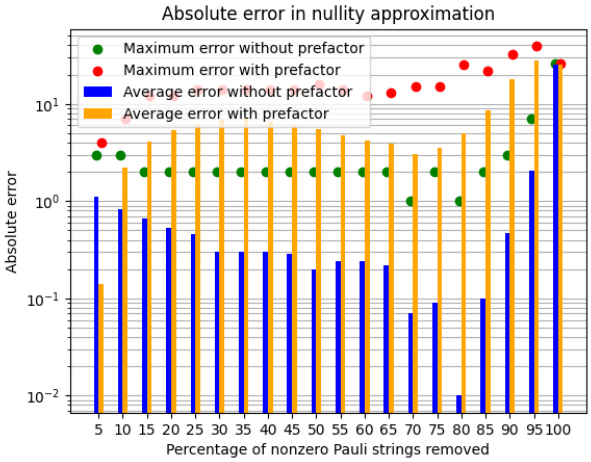


Figure 21: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.10

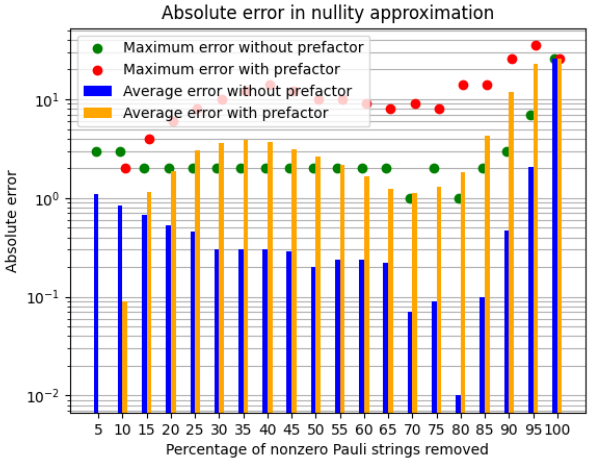


Figure 22: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.15

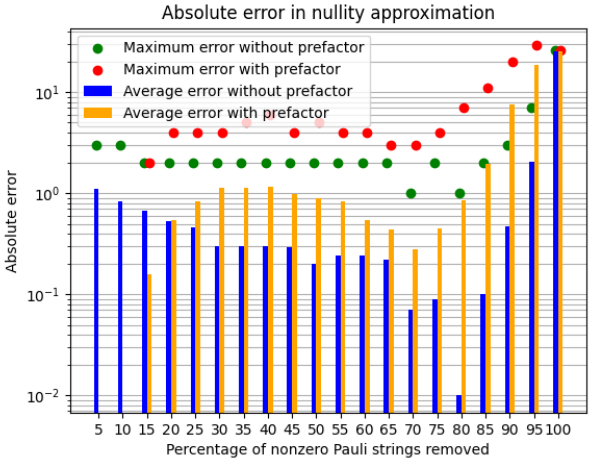


Figure 23: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.20

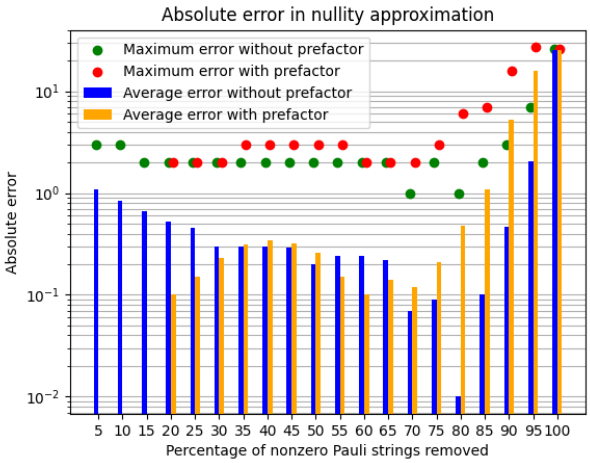


Figure 24: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.25

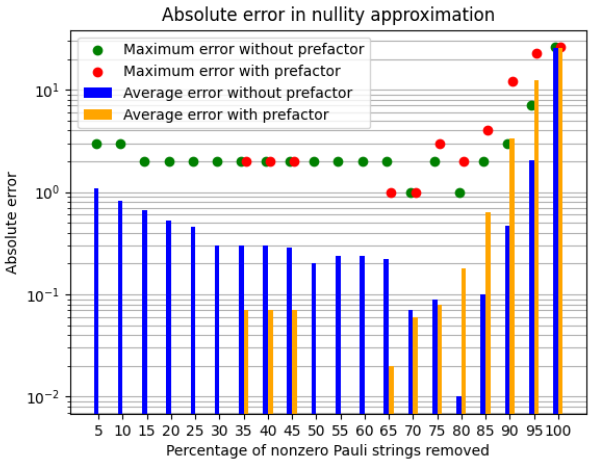


Figure 25: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.30

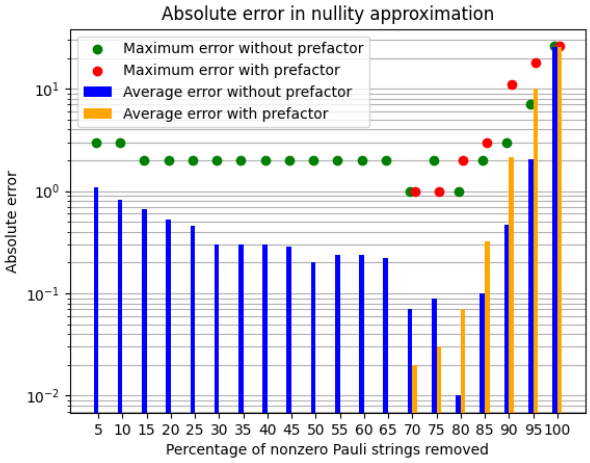


Figure 26: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.35

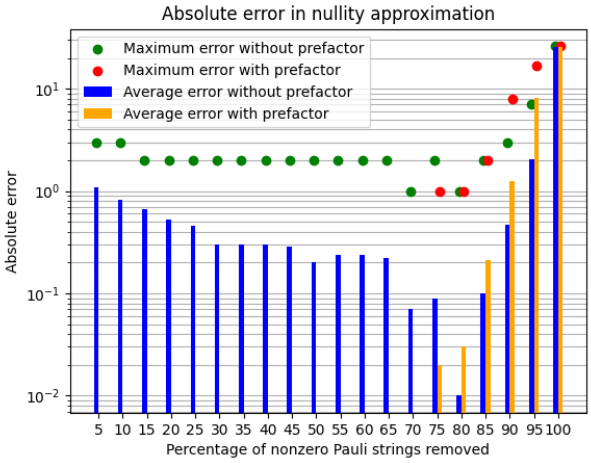


Figure 27: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.40

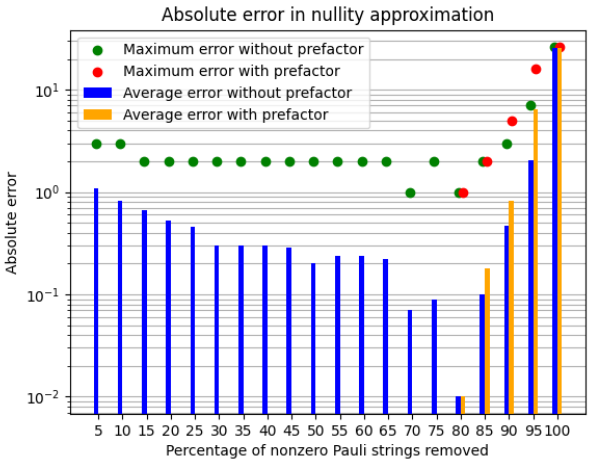


Figure 28: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.45

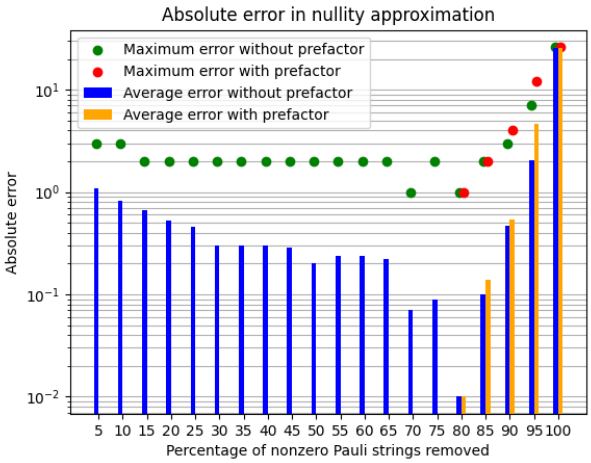


Figure 29: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.50

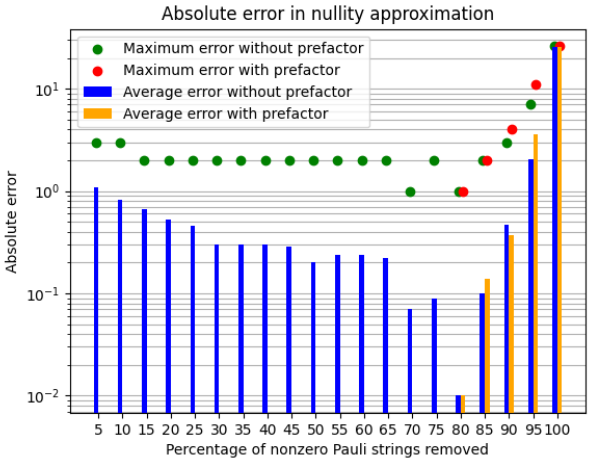


Figure 30: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.55

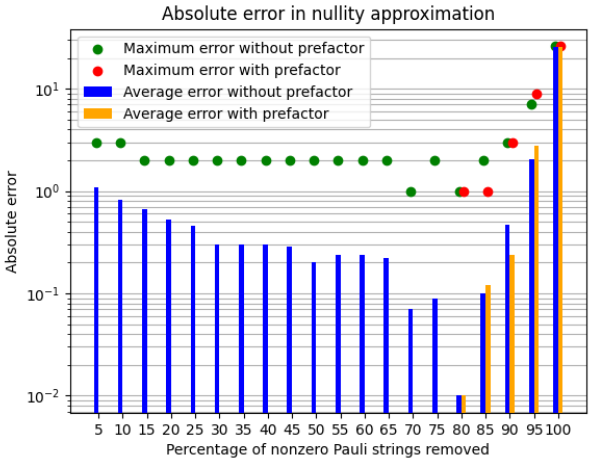


Figure 31: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.60

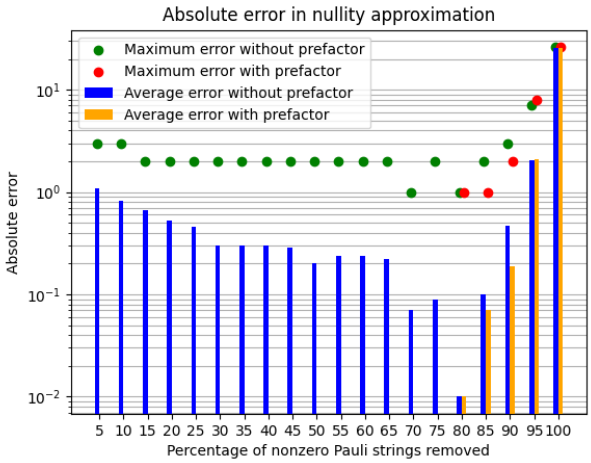


Figure 32: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.65

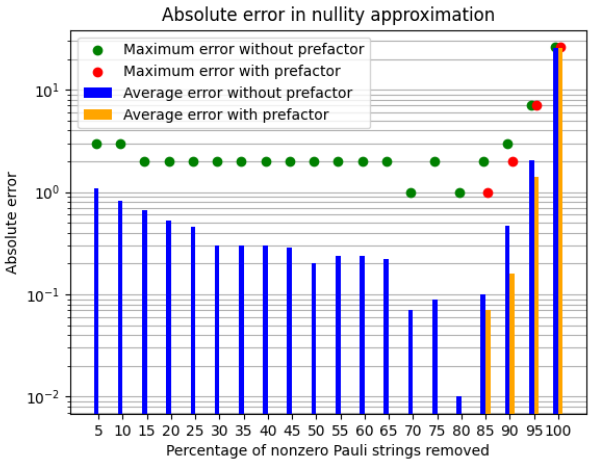


Figure 33: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.70

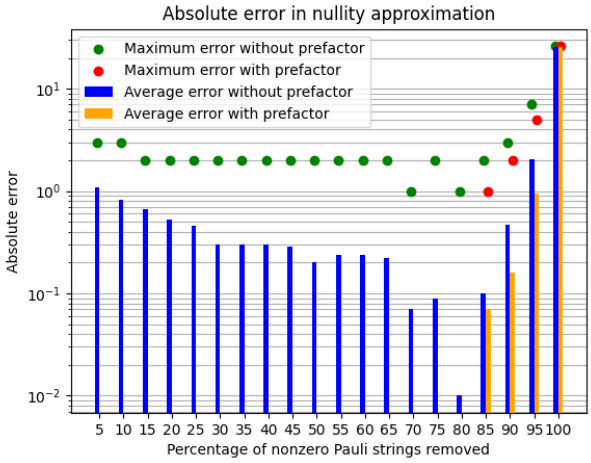


Figure 34: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.75

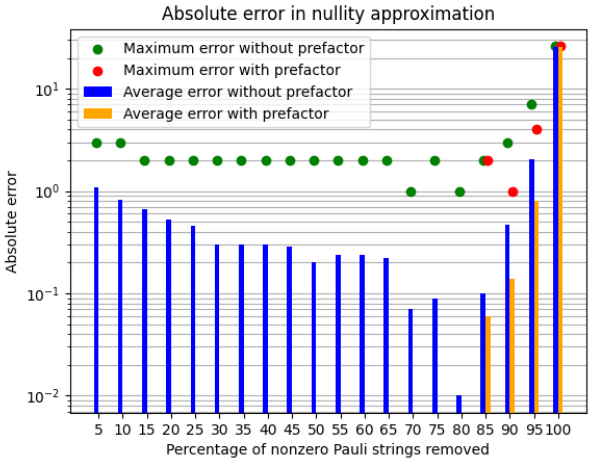


Figure 35: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.80

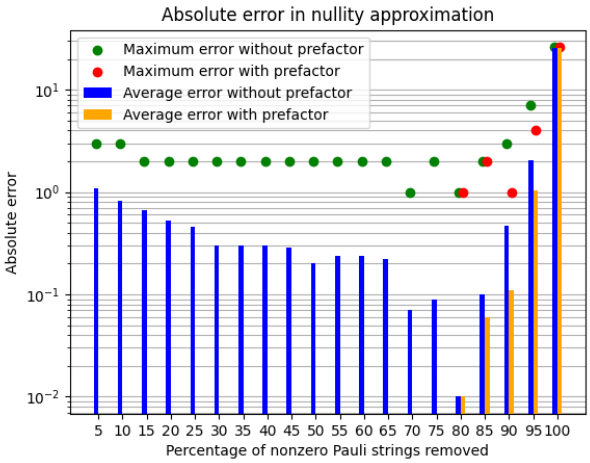


Figure 36: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.85

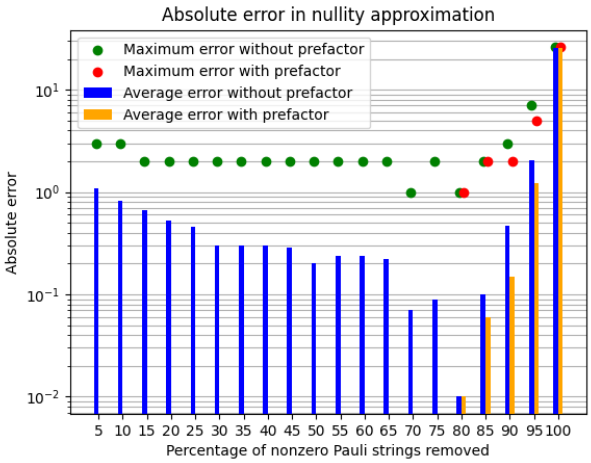


Figure 37: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.90

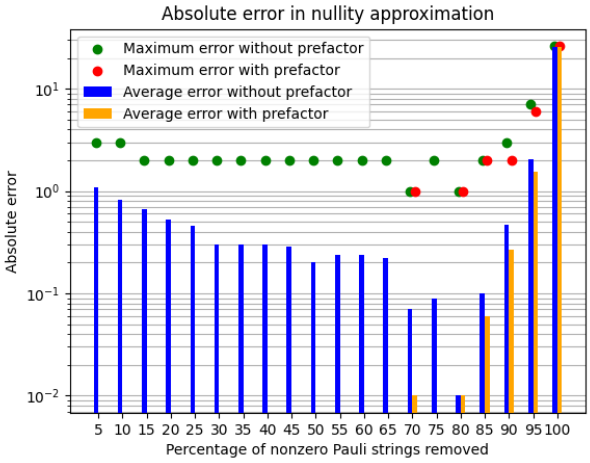


Figure 38: Absolute error for graph size $n = 7$ and clique size $k = 3$ with a prefactor of 0.95