Building a Markov Chain Monte Carlo Algorithm to Study the Entanglement Entropy of Quantum Hall States

Benjamin Jaderberg

Supervised by Dr. Frank Kruger
& Prof. Andrew Green

MSci Physics

Department of Physics & Astronomy
University College London

March 27, 2017

Abstract

The Renyi entanglement entropy (S_2) of a bipartite system is a measure of the entanglement between two groups of particles. We build, from scratch, a Markov chain Monte Carlo algorithm in C++ capable of calculating S_2 for a given trial wave function. The algorithm is verified through analytic and numerical calculations on the free-fermion wave function and then applied to the Laughlin states that describe the quantum Hall effect. For the nominal filling fraction $\nu = 1$ we construct a regime to extract the topological entanglement entropy γ and calculate its value to be $\gamma = 0.0878 \pm 0.018$. Afterwards, we explore for what reasons this differs from the field theoretic value $\gamma = 0$ and suggest solutions to improve the accuracy of our results going forward.

Contents

1 Introduction

In the context of modern Physics, our grasp of quantum entanglement is a monument to the success of quantum mechanics over a century after its formulation. Indeed, current research into applications of entanglement threatens to provide breakthroughs in a whole host of areas; condensed matter physics, atomic physics and cosmology to name a few. Furthermore, entanglement is essential in our best current models of building a quantum computer, and it is in these capacities that this project aims to contribute towards the field or research.

It is necessary to clarify that in the quantum realm, the state of a particle can often differ from the classical viewpoint due to quantum superposition. This means instead of always having a physical value, a quantum state exists as a combination of all possible states with their associated probabilities. Importantly, if we try to make a measurement on this quantum object, this superposition collapses into a single value.

A good example of this is the basic unit of information in a computer, called a bit. A bit is a binary object that can take on the value of either 0 or 1. Its equivalent in quantum computing is called a qubit, and a qubit is additionally able to exist as a superposition (combination) of both 0 and 1 until it is measured. As a loose analogy, we could think of a coin with two states heads and tails. Leaving the coin on a table, we could easily assign a value of heads or tails to it depending on which side was facing up. However, if we were to flip the coin it would be like a qubit – whilst up in the air its value could be described as an uncertain combination of heads and tails. This is true even though when the coin does land, it will still measure only heads or tails.

Practically we can represent qubits using the binary nature of spin in particles, where the spin can be either pointing up, down, or be in a superposition of up and down. With this understood, we can now address the key part of this project that is entanglement.

Suppose we create a system of two spins that we call A and B and impose that the total spin in the system is always zero. It is clear to see that if we to take a measurement of one spin, the other will always have an opposite spin in order to preserve the rule that total spin is zero. However as mentioned before, each of these spins individually exists as a superposition of up and down states until the moment of measurement. Indeed quantum mechanics makes it clear that it is impossible to know the result of such a measurement and it is truly random if a spin will measure as spin up or down. Therefore, a measurement of one spin must instantaneously somehow communicate to the other to collapse from superposition into the opposite spin. This process can happen over any distance and is faster than light – an example of the perplexing properties of entangled states.

Whilst this appeared paradoxical and deeply worrying to Einstein and others in 1935 [1, 2], we can now describe the entanglement of our spins A and B as:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle - |\downarrow\uparrow\rangle) \tag{1}$$

Where $|\psi\rangle$ is the wave function of the bell pair and $\uparrow, \downarrow$ represent the up and down spin eigenstates respectively. This is an example of quantum entanglement and is one of the 4 possible states that two spins can take, known as Bell states.

1.1 Entanglement Entropy in Condensed Matter Systems

Applying these concepts to condensed matter systems, it is important to note that entanglement can occur between more than just two particles, but rather between any two sets of particles. In the case of real materials, this bulk entanglement will normally involve $\sim 10^{23}$ particles and performing a computation over every pair would quickly reach the limit of modern computing power. Instead, the system can be partitioned and in this project we separate a d=2 dimensional system into a real space bipartition as seen in Fig.1. In this scenario, A and B are two subspaces of the total system[1]. However, in related work there is a plethora of ongoing research into other systems, including d=1 dimensions [3, 4], disjoint intervals instead of a partition [5, 6] and partitioning based on spatial [7, 8] or orbital momentum [9].

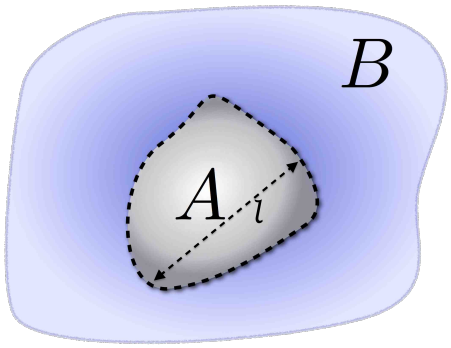


Figure 1: Graphical representation of bipartition in a quantum system S. S is split into subspaces A $\subset$ S with linear dimension l and B $\subset$ S, whereby A $\cup$ B = S.

In order to further analyse such an entangled system, one approach is to study the entanglement entropy. Traditionally, entropy is associated with an uncertainty or lack of information in a system[10] – often described as a measure of randomness. However, in quantum mechanics the entanglement of two systems causes an entanglement entropy, which is unrelated to the idea of traditional entropy and exists even for a pure system with no uncertainty. To calculate the entanglement entropy, we firstly use the degrees of freedom across our full system S to generate a density matrix ρ, often used to describe a mixed quantum state. By tracing out the density matrix over subsystem B, we can create a reduced density matrix (RDM) for subsystem A:

$$\rho_A = \mathrm{Tr}_B \rho \tag{2}$$

Subsequently, we can use our RDM to calculate the entanglement entropy. One such measure is the Von Neumann entanglement entropy, which can be calculated as such:

$$S(\rho_A) = -\mathrm{Tr}[\rho_A \ln \rho_A] \tag{3}$$

The Von Neumann entropy is however only one measure and is in fact part of a family whose generalisation is called the Renyi entanglement entropy. The general formula is given by:

$$S_\alpha(A) = (\frac{1}{1-\alpha})\ \ln \mathrm{Tr}\rho_A^\alpha \tag{4}$$

[1]For notation purposes, quantities of a particular subspace will have a corresponding subscript e.g. ρ_A refers to subspace A

For completeness it can be shown that this equation resolves back to the Von Neumann entanglement entropy in the limit $\alpha \to 1$ [11]. In this project, calculations are performed to determine the Renyi entropy for $\alpha = 2$, simply referred to as the first Renyi entropy or S_2. [2]

1.2 Spatial Scaling of Entanglement Entropy

One of the most intriguing results of calculating the entanglement entropy was discovered first by Bombelli et al. [12] in 1986 during a study of the entropy of black holes. From this work and many others, it has been shown that for a bosonic system with d dimensions, the entanglement entropy scales with the boundary of the partition. This is most comfortably explained in everyday 3 dimensional space: the entanglement entropy of a 3d system scales with the surface area of system, A. For this reason it is known as the *area law*, however this law would still apply for cases such as Fig.1. Here the entanglement entropy would scale with the perimeter of A, which itself would depend on the length l and a prefactor (e.g. if A were circular the boundary might be $2\pi l$). Generally then, for a system in d dimensions with linear dimension l [13]:

$$S_\alpha(A) \sim a_\alpha l^{d-1} \tag{5}$$

However, if our bipartite system was made up of fermions instead, it does not obey the area law as strictly. Instead, a logarithmic factor is introduced to describe how the fermionic case scales faster [14, 15]:

$$S_\alpha(A) \sim l^{d-1} \ln(l) \tag{6}$$

This deviation from the standard area law is a direct consequence of the presence of a well defined Fermi surface. This is further reinforced with several studies, most recently by M. Pouranvari et al [16], which shows that a weak random potential restores a strict area law. This is in agreement of the well defined Fermi surface theory – as the random potential effectively smears the boundary of the Fermi surface. Additionally, Equation 6 is not a comprehensive result but rather a specific conclusion based on the construction of the system similar to that shown in Fig.1. Whilst the phase of the fermionic matter has been so far unspecified, different phases have been shown to require other corrective factors, particularly at phase transitions which we talk more about below [13]. Furthermore, if the boundary between subspaces A and B was non-smooth, the corners of the boundary need to be factored in as another correction [13].

The increased range entanglement entropy for a fermionic system is interesting because the same result actually holds true for interacting fermions in the case of a Fermi liquid. This is surprising because a Fermi liquid does not have a well defined Fermi surface, an exception to our prior idea of why the logarithmic factor occurs. Recently there has been significant academic interest in quantum phase transitions; bringing Fermi liquids to what is known as the quantum critical point [17]. This is a phase transition that occurs for a Fermi liquid at absolute zero and so far it has been found that materials at this transition exhibit a variety of unexpected behaviours. Therefore, ongoing research into the entanglement entropy at quantum phase transitions is suspected and eagerly anticipated to produce interesting results.

[2] This is also sometimes referred to as the collision entropy

1.3 Project Overview

In the first section of this report, we have broadly introduced the theory behind quantum entanglement as well as entanglement entropy. We also examined the motivations behind our research as well as its possible applications in other fields.

In the second section, we solve the entanglement entropy of a N-particle free fermion system in closed form. This provides an analytic framework and an understanding of the properties of entanglement entropy for simply interacting systems.

In Section 3 we create a Quantum Monte Carlo algorithm to solve the same system calculation as in Section 2, but with a numerical approach. This is achieved through developing a thorough understanding of two key processes: Monte Carlo methods and the well established Metropolis-Hastings algorithm. After exploring some of the technical detail and challenges of simulating particles in computer code, we directly compare our two sets of results to prove the accuracy of the assembled QMC algorithm.

In the final part of this report we reach the exciting pinnacle of our work. Having built and verified our S_2 solving computer program, we apply it to a system that is extremely difficult to solve analytically – specifically the quantum Hall Effect. By studying the physical realisations of this phenomenon, we construct a regime that allows us to propose how the entanglement entropy scales. Importantly, we choose an implementation that does not directly copy any of the current literature and attempt to compare it to similar results at the forefront of academic research.

2 Entanglement Entropy of Free Fermions in Two Dimensions

The first part of the project involved analytically calculating the entanglement entropy for a simple system. We use the term analytic to highlight that the solutions in this section use closed form mathematics to produce an exact result, unlike the numerical approach in the next section. Therefore the results in this chapter could have been achieved using only hand-written maths, but in reality a computer program was built to dramatically speed up large calculations.

The motivation of this calculation was to produce a set of accurate results. This could then be used to directly test the accuracy of the algorithm we hoped to build, since both methods should produce the same values.

2.1 Wave Function of Free Fermions

The system being solved in this section is similar to the one seen in Fig.1, however with non-smooth boundaries such that A and B are square boxes. Additionally, the system is then populated with N number of fermions that do not interact with one another (also known as free fermions).

Since all quantum mechanical systems are described by a wave function, the first step of this calculation is to find the wave function for a single free particle. This can be derived starting from the Hamiltonian of a free fermion $\hat{H} = -\frac{\hbar^2}{2m}\nabla^2$, where ∇ is the del operator, m is the mass of the fermion and $\hbar$ is the reduced Planck constant. Substituting this into the Time Independent Schrdinger Equation (TISE) for 2 dimensions, we arrive at the wave function:

$$\psi = \frac{1}{L} e^{i\mathbf{k}\cdot\mathbf{r}} \tag{7}$$

Where $\mathbf{r}$ is the particle's position in x-y Cartesian coordinates, $\mathbf{k}$ is the momentum and the normalisation constant $\frac{1}{L}$ arises because our system is a square box of length L. Now we know the single fermion wave function, a general expression can be calculated for the wave function of a much larger N-particle system. This is achieved using the *Slater Determinant*:

$$\Psi(\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N) = \frac{1}{\sqrt{N!}} \begin{vmatrix} \psi_1(\mathbf{r}_1) & \psi_2(\mathbf{r}_1) & \cdots & \psi_N(\mathbf{r}_1) \\ \psi_1(\mathbf{r}_2) & \psi_2(\mathbf{r}_2) & \cdots & \psi_N(\mathbf{r}_2) \\ \vdots & ;\vdots & \ddots & \vdots \\ \psi_1(\mathbf{r}_N) & \psi_2(\mathbf{r}_N) & \cdots & \psi_N(\mathbf{r}_N) \end{vmatrix}$$

2.2 Momenta States in a Fermionic System

Studying the single particle wave function in Equation 7, there is an important practical consideration that needs to be made. Since the particles in our system are in fact fermions, they obey the *Pauli Exclusion Principle* (PEP) and therefore cannot occupy the same quantum state. By extension, the fermions in a many-body system can also not occupy the same momentum state and must have unique $\mathbf{k}$ values.

As an example, for a system of 37 particles in the ground state, the system is filled from the lowest energy level up. This corresponds to fermions arranging themselves into the next non-filled state closest to the origin, as shown by Fig.2. This particular number of particles happens

to be non-degenerate, such that the highest energy level forms a completely full shell. Studying systems with a non-degenerate number of particles, such as $n = 37$ before or $n = 5$, often simplifies our calculations. Therefore, the first task in this project was to produce a list of the particle numbers that result in non-degenerate shells and their respective shell radii. These numbers will continue to crop up throughout this report and the full list is plotted in Fig.3. As expected the number of particles scales as the area enclosed within the shell, however with clear discrete bumps. The results were calculated using an iterative program written in C++ that represents particles as points on a Cartesian grid, hence the often interchangeable use of the terms point and particle in much of this chapter.

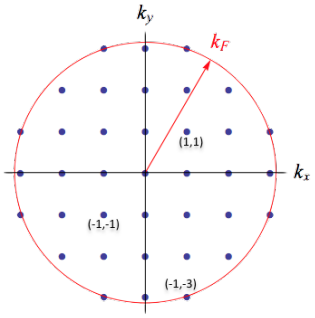


Figure 2: Filling of fermions in reciprocal space up to the Fermi surface. The quantisation of momentum states is a consequence of solving the TISE in the case of a free particle. Adapted from [18].

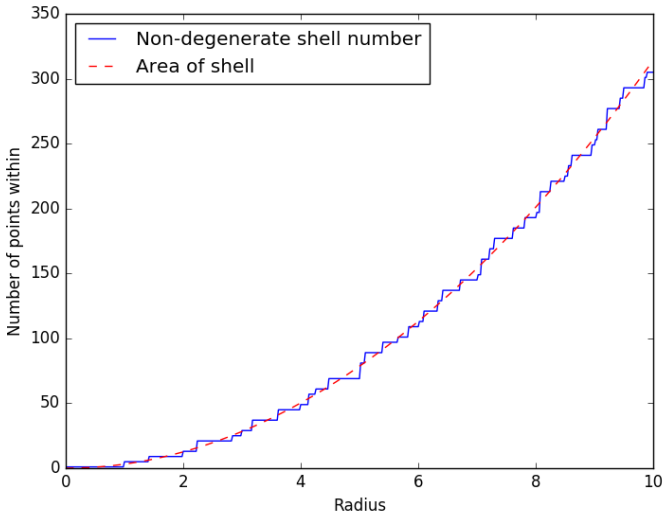


Figure 3: Graph to show the number of fermions in a non-degenerate shell of radius r. The radius refers to a circular shell in momentum-space for a ground state fermionic system.

2.3 Calculation of Renyi Entanglement Entropies

In previous sections we stated that calculating the entanglement entropy required manipulating a reduced density matrix (see Equation 4). However, creating the RDM can be quite a complex

and lengthy process itself. Instead, it was importantly proven by Calabrese et al.[19] that for a free fermion gas the entanglement entropy could be calculated with only knowledge of the individual wave functions. To do this one first has to construct an overlap matrix, which describes the overlapping of each particle's wave function with one another. Mathematically, this will be a matrix of size $N \times N$, with elements $A_{ij} = \langle \psi_i | \psi_j \rangle$. Expressing this Dirac notation as integrals and substituting in Equation 7:

$$A_{ij} = g(k_{i_x} - k_{j_x}) \cdot g(k_{i_y} - k_{j_y})$$

Where

$$g(k) = 1/L \int_{-l/2}^{l/2} e^{-ikx} dx$$

With some further manipulation we arrive at two possible forms of g(k):

$$g(k) = \begin{cases} \frac{1}{kL}(2sin(\frac{kl}{2})) & \text{if } k \neq 0 \\ l/L & \text{if } k = 0 \end{cases}$$

It is important one considers both cases because the diagonal elements in the matrix will be the overlap of one particle with itself. This means $k_i = k_j$ and as such the matrix element A_{ii} reduces to $g(0) \cdot g(0)$. Having constructed the overlap matrix we can solve it to yield a set of eigenvalues $\lambda_1 \ldots \lambda_N$. We can now calculate any of the Renyi entanglement entropies by substituting the eigenvalues into Equation 8. Note that similar to the original method, this equation resolves to one of a slightly different form for the Von Neumann entropy S_1.

$$S_\alpha = \sum_{i=1}^{N} \frac{1}{1-\alpha} \ln[\lambda^\alpha + (1-\lambda)^\alpha] \tag{8}$$

By adopting this approach, calculating the entanglement entropy for any number of particles is reduced to one key process of solving the eigenvalues of a N-dimensional matrix. Whilst this may appear difficult on paper, such a calculation is not computationally intensive even for a home computer up to approximately n$\sim$500. This allowed us to produce a large range of analytic results without requiring access to large computational resources.

2.4 Computation and Results

A computer program was written in C++ to follow the maths outlined in the previous subsections. Here we will briefly outline key characteristics of the code, since it also forms the basis for the programs seen later in this project.

A Point class was created (see Appendix B.1), defined with member variables x and y. All of the particles in this project can be expressed as points in two dimensional vector spaces, so it was anticipated that representing this as a custom object would save time overall. Furthermore, the subsequent analytical results are also plotted on two dimensional graphs. In this way the Point object also finds a second use as being a container for storing and exporting two dimensional data, even if it has no relation to physical particles.

By creating a solution in which Points represent particles, it is undoubtedly important that the Points can undergo the same transformations and contain the same properties as particles might in some coordinate system. Therefore, member functions were created to do the following coordinate and vector methods:

- Find the distance between two points
- Add or subtract two points
- Translate a point
- Create a copy of a point

With this in mind, a system containing several particles could then be implemented as a collection of Points objects. The system was then solved using the earlier analytic formulas, for which specific programming can be seen in Appendix B.2. Creating matrices and solving eigenvalues was done using the linear algebra C++ template library Eigen. Moreover, results were then exported to Python via a .txt file where they were plotted using the matplotlib package.

The full spectrum of analytic results are showcased with three graphs. Fig.4 plots the different measures of Renyi entropy against $Z = \frac{l}{L}$, where Z can be recognised as the ratio of the length of subsystem A to the length of subsystem B. Importantly, there is a distinct shape for how the entanglement entropy scales with Z; growing almost linearly until it is suppressed by the boundary conditions around Z=0.8. Knowing that entanglement only occurs between groups of particles in box A and box B, it seems obvious that there would be no entanglement at Z=0 and Z=1 since there is no inner subsystem in both cases.

Fig.5 plots a similar looking graph, except in this case we fix the type of Renyi entropy and instead look at different numbers of particles. In this graph we see that the entanglement entropy not only scales with Z, but also with an increasing number of particles in the system. This is in line with our expectations: more particles overall means more particles entangled between the subsystems, which means greater overall entanglement.

Fig.6 is perhaps the most significant result from the analytic calculations. Recalling and applying Equation 6 to two dimensions, we assert that $S_2 \sim l\ln(l)$. By considering this relation in terms of density ρ, number of particles N and system length L we can arrive with some algebraic manipulation at:

$$\frac{S_2}{\sqrt{N}} = a\ln\sqrt{N} + b$$

One can recognise that $\frac{S_2}{\sqrt{N}}$ is in fact a linear function of $\ln\sqrt{N}$. Plotting these quantities against one another, Fig.6 confirms that a straight line is produced. In doing so, the $l\ln(l)$ scaling of fermions was able to be independently verified.

In the next section of this report we attempt to simulate the same physics in an altogether different manner. These graphs provide the framework to rigorously test this, comparing not only the general shape and correlation but also the specific numerical values to our future attempts.

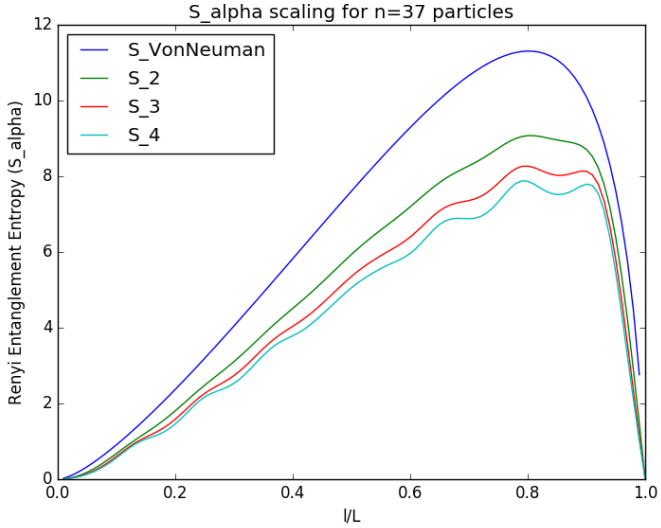


Figure 4: Graph to show the scaling of different Renyi entanglement entropies versus the size of the subsystem, for a fixed $n = 37$ number of particles.

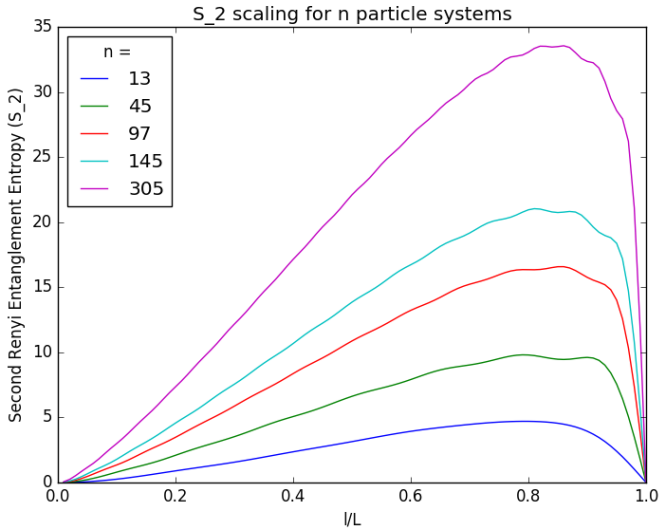


Figure 5: Graph to show the scaling of the Renyi entropy S_2 versus the size of the subsystem, for a varying number of particles.

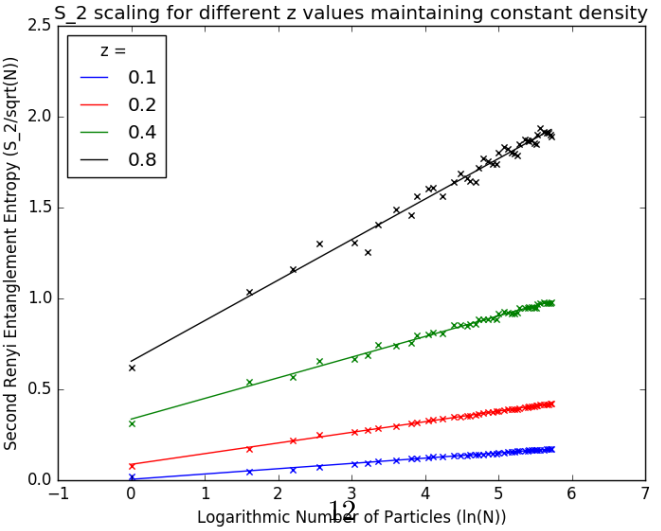


Figure 6: Graph showing $\frac{S_2}{\sqrt{N}}$ vs $\ln(N)$ for varying subsystem sizes at constant density. A least squares fit reveals the linear relationship, proving fermions obey a logarithmic correction to the area law.

3 Solving S2 Using a Quantum Monte Carlo Algorithm

The principal goal of this project was to build an algorithm capable of calculating S_2 for any system with varying general parameters. Whilst at first this may seem to have no obvious benefit compared to the analytic method, the impetus behind such an algorithm is that there are many complex systems in physics that cannot be solved in closed form. Changing properties of the system such as the size, shape, particle type, environment or temperature would all result in a problem so multifaceted that one would struggle to solve it with analytic maths. Instead, we have built an algorithm that takes a fundamentally numerical approach that is only limited by computational power.

3.1 The Monte Carlo Method

The underlying procedure behind the algorithm built in this work is a particular class of statistical algorithms known as a Monte Carlo (MC) method. MC methods are used to incredible success over a whole range of research fields and in the context of various fields have many names. Most generally the methodology used in this project is known as Markov Chain Monte Carlo (MCMC), but for our specific application to quantum mechanics this is often referred to as quantum Monte Carlo (QMC). Regardless, all MC methods are rooted in their use of repeated random number sampling to solve a problem from a numerical perspective.

A good example of how a MC method can be applied is solving the area under a graph curve. The analytic approach would be to apply traditional calculus – integrating the line between the upper and lower limits. On the other hand, a MC method would be to take a random point in the graph space and record whether it is below or above the curve. As we continue to repeat this hundreds of times, the ratio of the number of points below and above the curve would begin to converge to the ratio of the areas below and above the curve. This allows us to approximate the integral of the function f(x) without any knowledge of the equation that describes it. Fig.7 shows what such an example may look like after a few iterations.

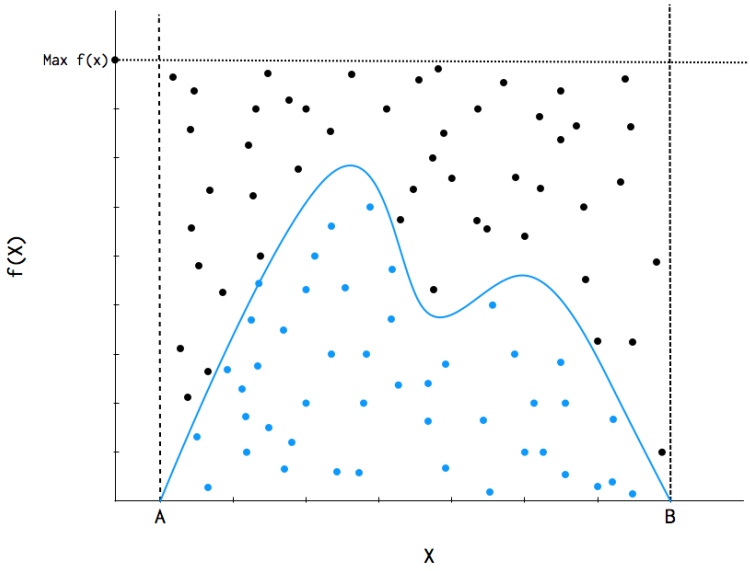


Figure 7: An example of MC integration. Counting the number of points below the curve as a ratio of the total points will converge on the ratio of the integral to the total area of the graph.

3.2 Building the Algorithm

It is important to remember that the QMC algorithm built in this project needs to be validated by comparison to the analytic results. Thus, an important starting point in creating the algorithm was to simulate a system identical to the one used in Section 2. Once again, this system is a square box of length L, with a bipartition creating a smaller box of length l, filled with non-interacting fermions.

To recreate this set-up, we initialised the algorithm with a number of Point objects on a two dimensional grid. Furthermore, because these Points represent particles of wave function $\psi = \frac{1}{L}e^{i\mathbf{kr}}$ they need to be initialised with appropriate $\mathbf{k}$ and $\mathbf{r}$ properties. For N particles, assigning the $\mathbf{k}$ values is trivial and follows the filling of unique energy levels up to the Fermi surface as discussed earlier. However, there is no such restriction on the spatial distribution of particles which are allowed to be found anywhere in the box. To reflect this, the system was initialised with particles in random positions given by $\mathbf{r}_i = \binom{x_i}{y_i}$ where $x_i, y_i \in [-\frac{L}{2}, \frac{L}{2}]$.

Random numbers are instrumental to all MC methods and therefore it is important we generate them correctly. For our algorithm in C++, a global random device was used which generated random numbers according to the Mersenne Twister algorithm. Specifically, the MT19937 version used is very well suited to a MC method as it will only repeat itself after $2^{19937} - 1$ numbers. By comparison, some older random number generators have a period of $\sim 2^{32}$, which could easily be less than the number of iterations required in some MC processes.

After initialising the particles in a random configuration, the task still remains to write a MC method that will calculate S_2. To do this, we were guided by the scheme initially laid out by Hastings et al.[20] (as were a variety of other works[18, 21]) and it is strongly suggested to refer to this paper for further details on the origin of equations that are quoted below. In Hastings' scheme, through clever manipulation it is shown that S_2 can be evaluated via a different approach to Section 2. By considering two copies of the system, R and R', the entanglement entropy can be calculated as:

$$e^{-S_2} = \langle \Psi | \, SWAP_A \, | \Psi \rangle$$

Where the right hand side (RHS) of the equation is the expectation value of the SWAP operator applied on subsystems A and A'. Fig.8 provides some further clarification on the system we are working in. Whilst this equation is technically an integral, it was shown before that any integral can be solved by a MC method if it can be deconstructed into a sum over many points. Doing this whilst simultaneously applying some further manipulation on the RHS of the equation, we arrive at the slightly more revealing form of Equation 9:

$$e^{-S_2} = \sum_{\substack{a,b \\ a',b'}} |\Psi(a,b)|^2 \cdot |\Psi(a',b')|^2 \cdot \frac{\Psi(a,b')\Psi(a',b)}{\Psi(a,b)\Psi(a',b')} \tag{9}$$

Where a, b represent the particles contained within boxes A and B respectively. From the previous explanation in section 3.1, we can see how a Monte Carlo method might be implemented to solve such an equation:

1. Generate two boxes each with all particles placed in random positions.
2. For this particular configuration, calculate the RHS of the equation.
3. Iterate (repeat) the last two steps, each time summing the value of the RHS.

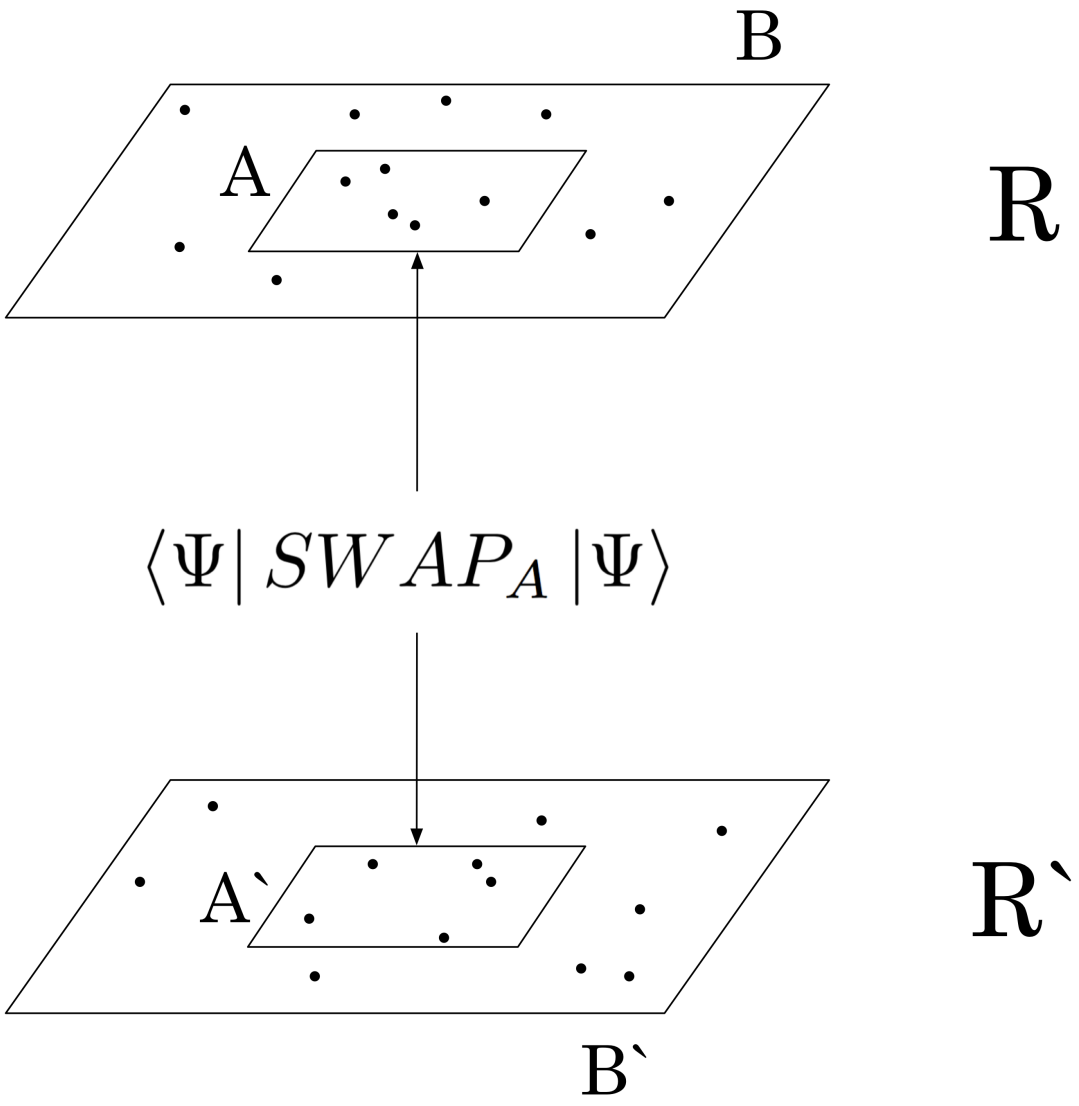


Figure 8: A diagrammatic representation of the scheme used by our QMC algorithm. A copy of the system R' is created from which the SWAP operator can be applied to the inner subsystems A and A'. Swapping these particles is a key step in calculating S_2 for the original system.

4. As the number of iterations approaches infinity, we will have calculated a value for every possible configuration. At this point, our answer converges on the exact solution to e^{-S_2}.

Whilst this showcases the brilliance of Monte Carlo methods, in real life we are unfortunately unable to carry out infinite iterations. Limited by both time and computational power, we may only be able to perform say one million iterations. In this case our MC method produces only an *approximate* solution, because there is a chance our random configurations did not cover every possible state.

In this project, we use a mathematical trick to improve the effectiveness of the MC method. Looking at the summation in Equation 9, one notices that the first two terms are simply the probabilities of finding each system in that particular spatial configuration. Significantly, it can be said that each term is in fact taken from a larger probability density function (pdf) which describes the likelihood of any configuration. If one were to have such functions, we could generate states according to the pdf of the system, rather than generating new random configurations every iteration.

The benefits of this trick are twofold. Firstly, by sampling from the pdf we are iterating over states that are more likely to physically occur in the system. This reduces the number of iterations required before we have seen almost configuration likely to occur. Secondly, by generating each box according to its pdf, the first two terms in Equation 9 are already accounted for. This means that each iteration performs a less intensive calculation and improves the speed of our algorithm.

This mathematical trick to improve the efficiency of our MC method relies on knowing the pdf of the free-fermion wave function, which we don't yet. Incorporating the crucial step of calculating what the unknown pdf looks like, and then sampling a MC method from that, is achieved using a procedure known as the Metropolis-Hastings algorithm.

3.3 Applying the Metropolis-Hastings Algorithm

Developed by Nicholas Metropolis in 1953 and then extended by Wilfred Hastings in 1970, the Metropolis-Hastings (MH) algorithm is a MC method that takes its Monte Carlo sampling from a probability distribution. What makes this algorithm so useful is that we do not even need to know the full probability distribution to apply it. Instead, we only require a mechanism to calculate the probability of individual states. Given this, by continuously iterating the MH algorithm the true pdf is eventually converged on, as an approximation based on the number of iterations. The exact steps of how the MH algorithm works is shown in the first part of the flowchart Fig.9. Written out more generally here:

1. Initialise the system in accordance with the best available prior knowledge of what the pdf might look like (known as the prior).
2. Calculate the probability of this particular configuration of particles using $P = |\Psi|^2$.
3. Propose moving every particle in the system a small distance dr in a random direction
4. Calculate the probability of this new configuration P_{new}.
5. If $P_{new} > P$ then accept the proposed move. If instead the new configuration is less likely to occur, accept it with probability $\frac{P_{new}}{P}$.
6. Carry out desired MC method for this sampled configuration.

7. Repeat from step 3.

After delving into the details of how the Metropolis-Hastings algorithm works, it is easy to see how configurations with greater probability will be sampled more often than less likely ones. Crucially, the number of iterations required before the algorithm is sampling according to the true pdf depends on how close our prior was to the actual solution. Therefore if our prior is an extremely good educated guess, the algorithm will produce more accurate results in a shorter time when compared to a completely ignorant prior. In the case of free-fermions, our prior is to start with the particles randomly populated across the boxes, derived from our understanding that the wave function is evenly distributed with no preferred spatial point.

One of the most important concepts in using the Metropolis algorithm[3] can be uncovered by inspecting the effect of the term dr in step 3. Bearing in mind that the purpose of the algorithm is to converge on the most likely states of a system, lets suppose we make dr very large compared to the system size. The benefit of this is that we will generate configurations spanning over the whole system, such that within a small number of iterations we will have recorded what the wave function approximately looks like everywhere. However this is also exactly the problem; setting dr too large can overlook the finer detail in a system or even fail to converge on the true pdf altogether. Every time the algorithm iterates onto a state of high probability, the next iteration will propose to move all the particles far away again. Saying for example this new state has a probability of 10 times less likely, one of two things can happen here. Either the new system is accepted 10% of the time and we have moved away from where most of the wave function lies, or the new system is rejected, no calculation is executed and we propose a new move. This in itself is a problem, because if 90% of iterations fail then it will take a very long time to produce any results.

On the other end of the spectrum, if dr is set too small we encounter an entirely different problem. It is plausible that even over several million iterations the particles may not travel very far from their random starting place. Ergo, there is a risk of not analysing the entire wave function and instead producing an incomplete result without knowing so. Furthermore, depending on the specific wave function there is also the possibility of becoming trapped in a local minima – unable to move particles far enough to escape a small but non-trivial spike in probability density.

For these reasons, initialising the value of dr for each individual run of the algorithm is very much a goldilocks scenario[4]. Analysing whether the value of dr used was just right or not can be interpreted from the acceptance rate of the run. The acceptance rate η is given by:

$$\eta = \frac{N_{acc}}{N}$$

where N represents the number of iterations carried out. Whilst it is commonly quoted that the optimal Metropolis acceptance rate is 0.234 [22], a variety of other works cast doubt on what is still a greatly debated question[23]. Due to the preliminary nature of the work in this project and the use of only personal computational resources, it was deemed that producing optimal acceptance rates was in fact not a large priority. Through empirical analysis it was determined that in fact any acceptance rate between 0.3-0.7 produced similarly accurate results.

[3]When dealing with symmetric distributions, there is a special case of the Metropolis-Hastings algorithm called simply the Metropolis algorithm.

[4]By which we mean the value of dr needs to be set not at either extreme, but within some acceptable middle range.

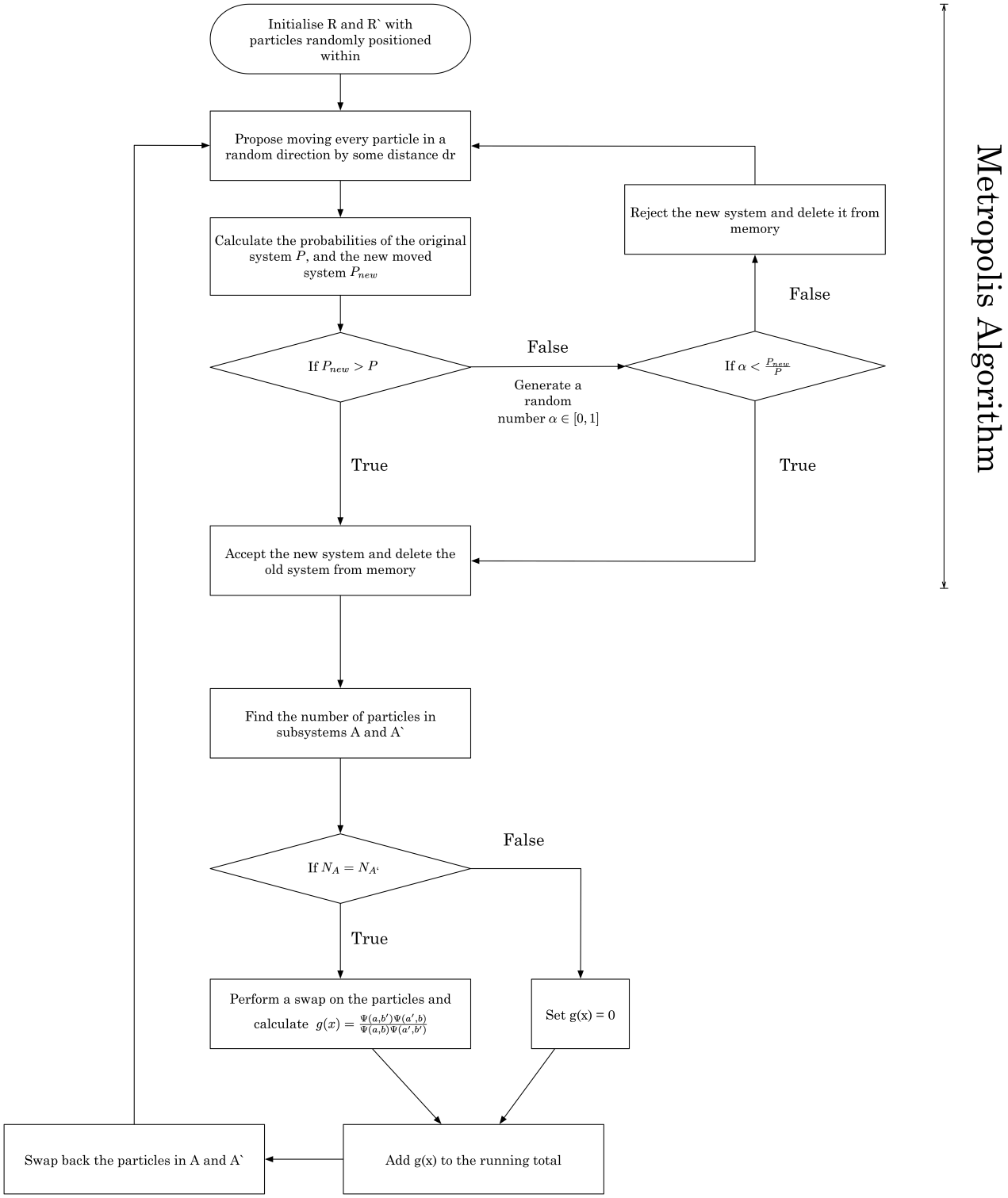


Figure 9: A flowchart the procedure of our QMC algorithm. Starting with a random system configuration, we apply N iterations of a Metropolis algorithm. When accepted, a swap is performed on the subsystems A and A' to then calculate the value g(x). Summing $g(x)$ and normalising by $n_{accepted}$ we arrive at a value for S_2.

Still, manually adjusting the value of dr to be acceptable for different systems proved to be a hindrance for the free-fermion case.

The Metropolis algorithm achieves the exact trick we hoped for to decrease the complexity of our calculation. Instead of using random sampling and a Monte Carlo method to calculate Equation 9, we can use Metropolis sampling followed by a Monte Carlo evaluation of the new reduced Equation 10:

$$e^{-S_2} = \frac{1}{N_{acc}} \sum_{i=1}^{n} g(x_i) \tag{10}$$

Where x_i is the i'th configuration, n_{acc} is the number of Metropolis accepted moves and $g(x) = \frac{\Psi(a,b')\Psi(a',b)}{\Psi(a,b)\Psi(a',b')}$ is the remaining term from before. Studying $g(x)$ closer, we can identify it is a product of the wave functions after subsystems A and A' are swapped, divided by a product of the original wave functions. Therefore, our algorithm can evaluate $g(x)$ by simulating swapping the particles and then recalculating the wave functions. Combining both the Metropolis method and this calculation of $g(x)$ encapsulates the entire structure of the QMC algorithm that was constructed in this project. Visually, the algorithm is laid out in the flowchart Fig.9 and the source code is available to read in Appendix B.3.

3.4 Results

After creating the QMC algorithm, it was important to test that it was working as intended. For a given number of particles, the algorithm was adjusted so that it produced a range of S_2 values from Z=0 to Z=1 in increments of 0.1. Repeating these runs for different numbers of particles, a set of results was produced mapping S_2 against Z for up to $n = 25$ particles. This is noticeably less particles than studied in the analytic section, due to the MC simulation being far more computationally intensive than solving one matrix.

A comparison of the analytic and QMC results can be seen in Fig.10. Each scatter point on this graph is generated using one million iterations of the QMC algorithm.

The error bars of the QMC method were generated using the standard results for the uncertainty in MC data:

$$\Delta I = \frac{\sqrt{g^2(x) - g(x)^2}}{\sqrt{N_{acc}}}$$

Where I is a label for the integral, which is the result of the sum of $g(x)$ in Equation 10. One can also recognise the numerator of this equation to be the standard deviation σ of our integral. This was then propagated through to the uncertainty in S_2 using:

$$\Delta S_2 = \Delta I / I$$

Looking at the results it is apparent that our QMC data is in strong agreement with the analytic data. We can see that whilst the error bars are very small for $n = 5$ and $n = 13$, the algorithm calculations match almost exactly the curve both numerically and in terms of shape. Furthermore, for $n = 25$ the QMC data lies on top of the curve to within an acceptable error. Failure of the algorithm to match finer structural details at around $Z = 0.8$ is a consequence

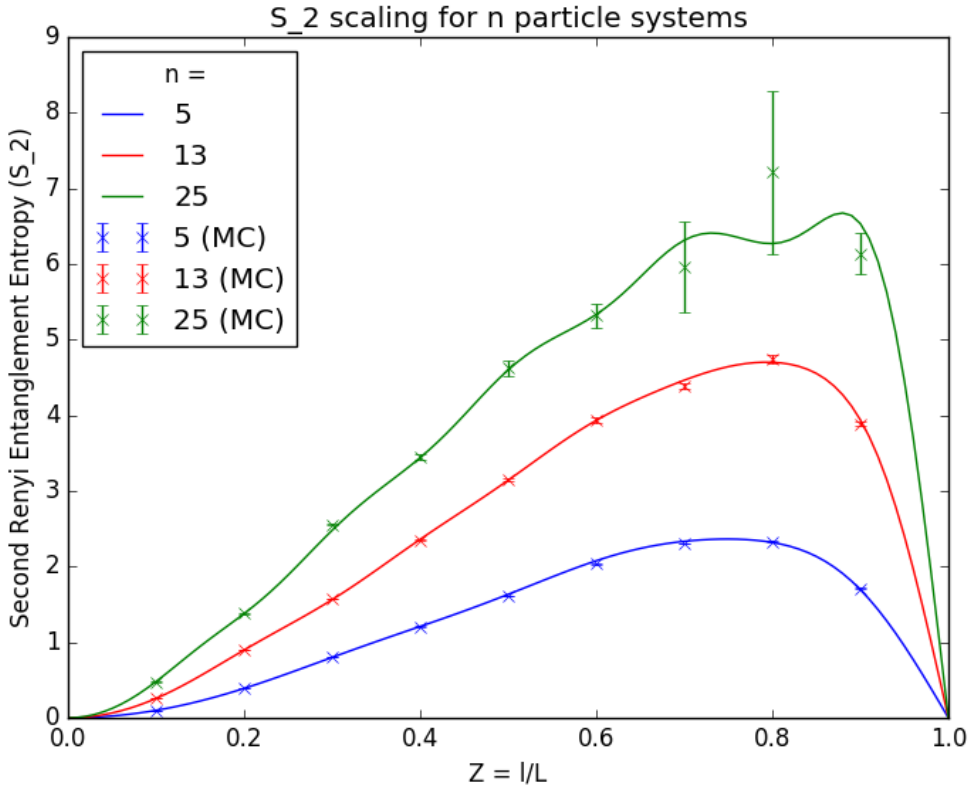


Figure 10: Graph showing the scaling of S_2 with subsystem size for different numbers of particles. The solid lines are the analytic results, whereas the crosses indicate results generated by our QMC algorithm.

of the increasingly large error bars at this limit. With more time or resources allocated, we hope to reduce the larger error bars by allowing the algorithm to run for many more than 10^6 iterations.

A keen observer may also note that for $Z = 0.9$ on the $n = 25$ line, the curve actually lies outside the error bar of the plotted point. However this does not cast doubt on equating the two results, since the error bars represent one standard deviation of the value. For methods involving randomly generated variables such as Monte Carlo, there is a non-negligible chance of producing data greater than one σ away from the true value.

This marks the end of the first part of the project. We have constructed a quantum Monte Carlo algorithm which calculates the entanglement entropy S_2 for the free-fermion wave function. Furthermore, the numerical calculations of the algorithm have been verified to match with the mathematical theory. Whilst this accomplishes the goals set out to achieve in the initial project brief, it is with great enthusiasm that the project was extended further. In the next section, we take the algorithm already built and apply it to particles far more exotic and colourful than free-fermions.

4 The Quantum Hall Effect

In this part of the project, we look at the novel application of our algorithm to the quantum Hall effect (QHE). The QHE is a particularly active and interesting area of research, with the 2016 Nobel Prize in physics partly awarded for advancements in our understanding of the QHE effect using topological concepts [24, 25, 26]. QHE systems result in an exotic and highly unusual type of matter, best described by an electron-fluid of sorts. Due to the non-trivial topological order seen in the this matter, we hoped that a study of its entanglement entropy could produce exciting results in an area of condensed matter research that has gained significant recent attention.

Throughout Section 4 there will be various references to the mathematical field of topology. Whilst a deep understanding of topology is not necessary to read and appreciate the work in this project, readers might still find it useful to briefly detour to Appendix A.

4.1 Background

A large part of the theory and understanding in this sub-section comes from the excellent lecture series by Cambridge's David Tong. Thus all of the equations and numerical facts quoted are in fact pre-emptively cited to him accordingly [27].

To observe the quantum Hall effect does not require a particularly complicated experiment. Firstly, get a number of electrons and contain them within a two-dimensional plane, for example on the surface of a material. By cooling the material to a low temperature and then applying a strong magnetic field, we observe a particularly strange result. We find that the Hall conductivity follows the equation:

$$\sigma_{xy} = \frac{e^2}{2\pi\hbar}\nu \tag{11}$$

Where ν was an unknown quantity originally measured to take only integer values. The absurd precision of these integers, measured to 9 decimal places, show that there is a true fundamental quantisation of σ occurring. This is known as the integer quantum Hall effect; it implies that as we increase the magnetic field on our system the conductivity will seemingly not change, but then suddenly jump up. Amazingly this effect is brilliantly clear, even with apparatus that is far too simple to see many traditional quantum effects.

Interesting as that may be, it is perhaps not so surprising that this relation is quantised. Quantum mechanics relies fundamentally on both the quantisation of light and of several features of the atom. One example is how the energy of charged particles following cyclotron orbits in magnetic fields are also quantised, into what are known as Landau levels. Furthermore, it has been shown that the quantised values of conductivity directly correspond to number of filled Landau levels [27], solving the initial mystery. However, it was measured in 1982 that in fact ν could also take fractional values and was subsequently labelled the filling fraction. This discovery of the *fractional* Quantum hall effect is perhaps what is so truly remarkable. In this case, not only are the Landau levels fractionally filled, but the quantum Hall phase has fractional properties too. Whilst it is a fundamental truth that the natural charge of an electron is -1, in a fractional QHE system the bizarre state of matter can be measured to have a fractional charge such as -1/3 or -2/5.

This leaves us with two options. One is that the electron itself is breaking apart, but even our most exotic experiments tell us this can't be true. The other option is that there is a new type of particle at play here, one we call an anyon. Measured fractional statistics point towards anyons existing somewhere in between bosons and fermions, however it would be rash to suggest they could be a new fundamental particle. This is because anyons have only been observed and are only theoretically understood to exist in two-dimensions. Instead, it is thought that anyons represent a loophole in our understanding of how invariance under rotation applies from three to two dimensions. How this loophole arises is more specifically explained via a topological argument in Appendix A. The Laughlin wave function is an ansatz, a guess, at what the wave function of these anyons might be like. Through a great deal of consideration, in 1983 Robert Laughlin suggested a trial wave function that would best describe the QHE: [28]:

$$\Psi(z_i) = \prod_{i<j}(z_i - z_j)^m e^{-\sum_{i=1}^{n}|z_i|^2} \tag{12}$$

Where z_i is the position of the ith particle, converted from Cartesian coordinates (x_i, y_i) to a complex number $z_i = x_i + y_i i$, and m is the filling fraction $m = \frac{1}{\nu}$. As a reminder, the wave function is so important in quantum mechanics because it is directly related to the probability of what state the system will be measured in (the probability density function is given by $|\Psi|^2$).

Incredibly, experimental data has shown that this wave function is almost the exact description of the QHE, to within 1% of the measured results. Whether this will continue to hold as researchers simulate increasingly complicated systems remains to be seen. However, the Laughlin wave function provides an astoundingly solid foundation for understanding the quantum Hall effect and it is for that reason it is a great model to apply our QMC algorithm to.

4.2 Studying the Laughlin Wave Function

Taking a closer look at the Laughlin wave function, there are two immediate and important features to pick out. Firstly, the Laughlin wave function can be viewed as a struggle between two competing terms: the *product term* $\prod_{i<j}(z_i - z_j)^m$ and the *exponential term* $e^{-\sum_{i=1}^{n}|z_i|^2}$. The product term is a function of the distance between two particles, such that as particles z_i and z_j are brought closer and closer together this term eventually reaches 0. Therefore the product term not only ensures that no two particles can be on top of one another, but it acts as an overall outwards pressure that favours all particles to be as far away from each other as possible. On the other hand, the exponential term does the exact opposite and suppresses the probability of finding particles far away from the origin. Of course we need to define what far away means quantitatively – but it is still important that if the exponential term was left alone it would favour a configuration of all particles occupying the same space at the origin of our system. Thus, Laughlin's educated guess laid out a wave function that is a battleground between these two elements.

Secondly, the Laughlin wave function has an embedded description of the fundamental behaviour of fermions within it. In the product term $\prod_{i<j}(z_i - z_j)^m$ there is a very intriguing power of m, which Laughlin defined himself as being able to only take odd positive integers. Why? Because when m is an odd number the wave function becomes anti-symmetric under exchange. This means that if we swap the particles Z_i and Z_j around then overall a factor of -1 is introduced into Equation 12. Anti-symmetry is part of what defines a fermionic wave function, so the presence of it in Laughlin's wave function suggests that anyons can certainly behave like fermions – even if only fractionally. This is only further reinforced by the behaviour discussed before;

systems described by the Laughlin wave function cannot have two particles occupy the same spatial state. This is strikingly similar to another fundamental property of fermions, the Pauli Exclusion Principal.

Combining these two characteristics is certainly illuminating about the general behaviour of the Laughlin wave function. However, as mentioned before what we are still missing is the finer nuances of what the pdf actually looks like (and by extension what Ψ looks like). Understanding these specific details is going to be essential if we are to later construct a regime to measure both the entanglement entropy and various scaling of the wave function.

Luckily, finding the pdf of a system for which we only have the wave function is a problem that has already been solved – the Metropolis algorithm introduced in Section 3. Isolating the Metropolis algorithm out of our previously written code, we created a stand alone program to analyse the Laughlin states for the filling fraction $\nu = \frac{1}{m} = 1$. This was run over 10 million iterations, where for every new accepted configuration the positions of each particle were recorded and binned into radial shells. The number of particles found in each bin was then divided by either the shell radius to produce Fig. 11a or by the shell area to produce Fig. 11b.

Before we do any analysis on these results, it is important to justify the following idea: the term density profile is completely analogous to probability density distribution. Our Metropolis approach can be reconciled with quantum mechanics by recognising that each accepted random configuration is equivalent to a measurement on the wave function – collapsing all the possible positions of particles into definite values. Furthermore, taking several million iterations is the same as superimposing the results of several million measurements onto one map of the system. Suppose overall we now have a profile of 1 million measured particles positions and a certain area A contains 10,000 of such points; this means 1% of all measurements found a particle within A. Suddenly we are able to justify the mental leap that the density of particles in this profile is directly linked to the probability density of a single particle measurement. In other words, over many iterations the Metropolis algorithm maps out the full probability density function $|\Psi|^2$.

Whilst both graphs show the same data, it is perhaps Fig. 11b that provides the best clarity on what these results mean for the Laughlin wave function. For a system of n=2 particles, starting from the origin outwards there is a steadily increasing probability of where one might measure the particle. After reaching a region of maximum density a sharp drop off occurs, whose exponential-like shape reflects the exponential term in the Laughlin wave function.

Most interestingly however is how the density profile evolves with an increasing number of particles. Analysing the patterns occurring as we move towards $n = 20$ particles, we find the following observations:

- Around the origin, a plateau of constant density develops that becomes increasingly significant as we approach larger numbers
- The value of the density for this plateau decreases, but at a rate that is seemingly approaching a limit
- The bump occurring after the plateau diminishes in size
- The drop-off in density after the bump becomes flatter as a consequence of the lower starting height, but does not appear to change width

Extrapolating these patterns to larger system of say N$\rightarrow$1000 particles, one could imagine that the density profile will evolve to just become a flat line of constant density up to a maximum radius, at which point the density drops-off. A physical interpretation of this implies an object

that consists almost entirely of uniformly packed particles, resting in equilibrium between two opposing forces that want to pull them in opposite directions. This is intriguingly similar to how one might describe the internal structure of a neutron star: neutrons homogeneously packed together into a dense core by gravity but held in place by the outwards pressure from the Pauli exclusion principal. This is only possible because neutrons are fermions, and the similarities of neutron stars to our results provide yet another fragment of evidence as to why we might consider anyons to be in some part fermions.

4.3 Adapting and Improving the Algorithm

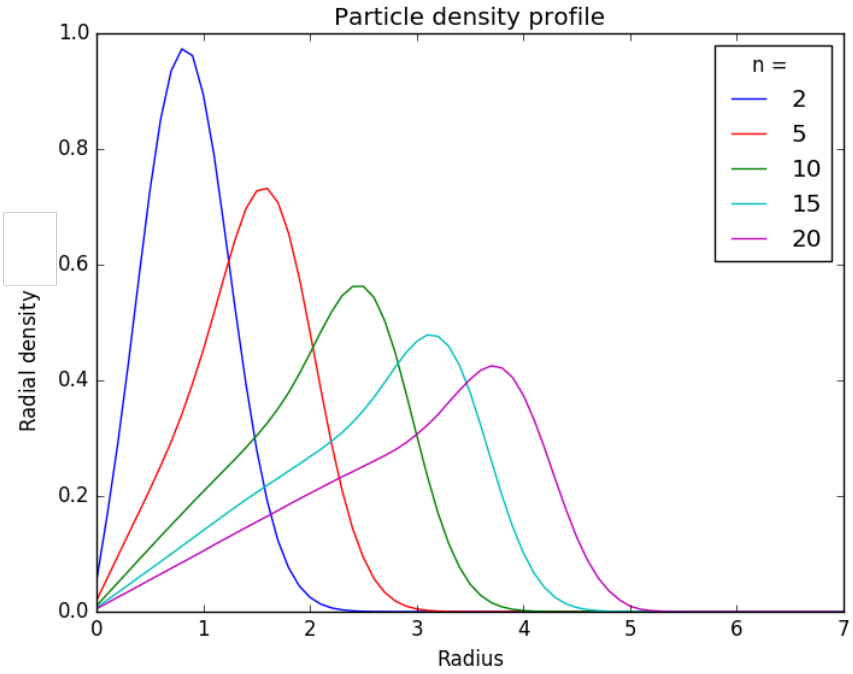
Despite our best attempts to construct a general algorithmic solution, it was always clear that solving for the QHE would require more work than simply substituting in the Laughlin wave function. Instead, there were several specific implementation differences to consider when compared to the free-fermion case. This also provided a natural opportunity to reflect on the processes already built and to ask whether they could be improved with regards to the two most important criteria – accuracy and speed.

One of the most important differences when comparing the QHE and free-fermion cases is perhaps the most obvious – how different their wave functions are. Whilst the free-fermion wave function has periodic features evenly distributed around the box it fills, the Laughlin wave function is not translationally invariant and instead shows distinct clustering based around the origin. Consequently, there is no clear reason as to why we should initialise the Metropolis algorithm with the same prior as the free-fermion case. Based on the previous analysis and plots of the Laughlin wave function, it was observed that the exponential term was dominant in describing at what radial distance the probability of finding a particle becomes negligible. If one takes the crude assumption that the Laughlin wave function is approximately only the exponential term, then it can be shown that 98.9% of all particles would be found within a circle of radius 3 standard deviations σ. A broad prior was constructed using this approximation, initialising all particles to be randomly distributed within a circle of radius 3σ. When calculated, $3\sigma = \frac{3}{\sqrt{2}} \approx 2.12$.

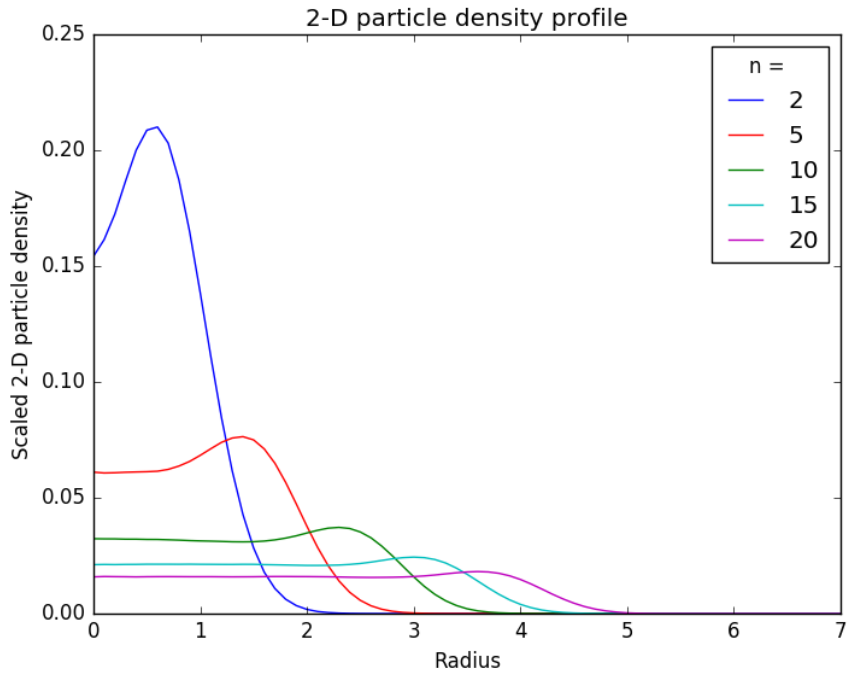
Another adaptation that was implemented to improve the algorithm's performance was specifically related to the SWAP calculation $g(x)$ in Equation 10, where $g(x) = \frac{\Psi(a,b')\Psi(a',b)}{\Psi(a,b)\Psi(a',b')}$. Through substitution of the Laughlin wave functions into this equation it can be shown that the exponential terms cancel out. Thus, doing such a calculation with the full wave function is a redundant additional complexity. The solution to this was to rewrite the `createWaveFunction()` method, adding an optional argument to return either just the product or the exponential term of the wave function by itself. For any numerical calculation, each term in a equation is its own source of noise. Hence, removing two terms from the $g(x)$ expression should in theory reduce the error in our calculation of S_2. The results shown in Table 1 provide evidence that implementing this adaptation reduced the error in the integral by at least a factor of two over the range of any number of iterations used in this project.

After considering the specific optimisations possible for the Laughlin wave function, thought was given into how the algorithm could be improved more generally.

As mentioned earlier, one of the only ways to measure whether the QMC algorithm both converged and covered the whole system is via the Metropolis acceptance rate. The acceptance rate is dependent not only on the iteration distance dr, but is also affected by the number of particles and the random statistics of what position the particles happen to be initialised in.



(a) Radial density profile. The results are normalised such that the area under each curve adds up to 1, corresponding to a definite probability of finding a particle within that region



(b) 2-dimensional density profile. This corresponds to Fig. 11a but with each density value divided by $2\pi r$

Figure 11: Graphs showing the density profiles of particles for a N-particle system described by the Laughlin wave function. This was achieved by allowing a box of such particles to evolve over 10 million iterations according to the Metropolis algorithm. At each accepted step, the radial position of all particles were binned into shells and recorded.

Number of QMC iterations (millions)	Δe^{-S_2}	
	Before exponential cancelling	After exponential cancelling
0.01	0.0451	0.00789
0.1	0.0129	0.00907
1	0.00300	0.00102
10	0.000738	0.000238
100	1.45e-05	3.98e-05

Table 1: Table showing the calculated error of the Monte Carlo integral, both before and after adapting the SWAP computation for the Laughlin wave function.

This had the rather tedious consequence of being unpredictable and requiring regular tweaking of the dr value to produce valid results. Therefore, one of the most glaring candidates for improvement was to automate this process – producing a less hands on and reliable algorithm. Additional code was written into the main method which adjusted dr based on the acceptance rate of the last 1000 iterations. Moreover, the adjustment to dr was made sure to be relative to the absolute value of dr and proportional to how extreme the acceptance rate is:

$$dr = \begin{cases} dr(\eta + 0.7) & \text{if } \eta < 0.3 \\ dr(\eta + 0.3) & \text{if } \eta > 0.7 \end{cases}$$

Test runs indicated this method could successfully fix dr values initialised up to 100 times from the goldilocks range in under 20 adjustments. For a QMC algorithm run 10 million times (as was done in the Laughlin case), that means only 0.2% of the entire simulation time was used adjusting to the correct values.

Another improvement to the QMC algorithm was the addition of a *burn-in* method. In the field of Markov chain Monte Carlo methods, a burn-in method refers to performing a number of iterations on the system but then throwing away any data collected. As normal the Metropolis algorithm samples states of the system, but instead of passing the data through to the S_2 calculation it is simply discarded. The motivation behind this is that the first configuration of particles initialised could be one that is extremely unlikely for the system to be physically found in, due to either a terrible prior or the bad luck of random initialisation. By discarding data for the first million iterations or so, we are able to converge towards the more likely system states without wasting time performing lengthy calculations on extremely unlikely configurations. Simply put, burn-in allows us to gain a better picture of the wave function faster.

The source code for the QMC algorithm as adapted for the Laughlin wave function can be seen in Appendix B.4.

4.4 Constructing a Regime to Measure the Area Law

Considering the aforementioned upgrades made to the QMC algorithm, one turns the attention back to the task of calculating the value of S_2 for QHE systems. Applying the same techniques as before, the task is to calculate the entanglement entropy via Equation 10 which as a reminder is:

$$e^{-S_2} = \frac{1}{N_{acc}} \sum_{i=1}^{n} g(x_i) \qquad\qquad g(x) = \frac{\Psi(a,b')\Psi(a',b)}{\Psi(a,b)\Psi(a',b')}$$

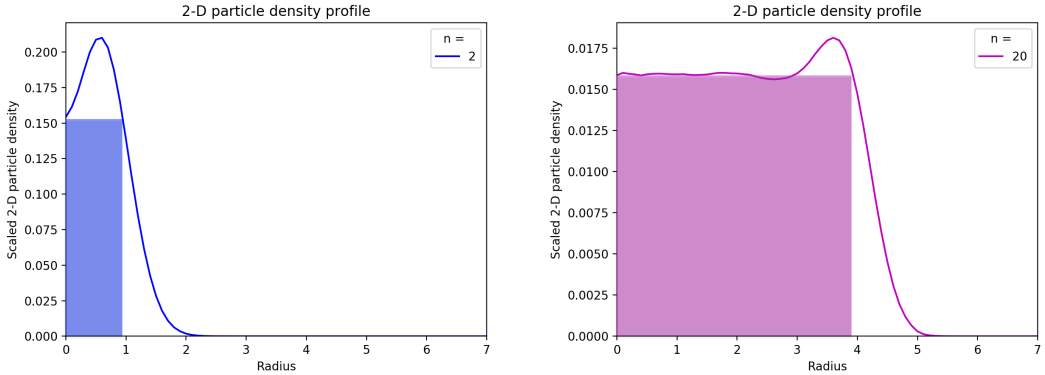
However, we are unable to define the subsystems A and B by the same bipartition seen in Fig. 1 because the Laughlin wave function is centred around the origin. Instead, A and B need to be two subsystems that cover the whole system, where the wave function behaves equally within and with an interface between them. An intuitive decision is to divide a box of length L in half by the vertical line $x = 0$ of length l, as illustrated in Fig.13f. Furthermore, from the observations in section 4.2 it is clear that the wave function does not take up all the available space but has a finite size. This means the outer box of size L is far less relevant than before, acting as mostly a container to the wave function by setting L to be of a relatively much larger size.

At this point having defined the subsystems, our QMC algorithm is let loose on the Laughlin wave function calculate S_2 for the range $n = 2$ to $n = 20$ particle systems. Yet without analytic test results to compare with, the difficulty remains in knowing whether any QMC results are correct – and if so then how to interpret them. To solve this, we propose that even the Laughlin states should obey one of the fundamental laws of entanglement entropy: the area law. Since the area law describes the scaling of S_2 vs l but our results describe S_2 vs n, the challenge is to find a way to link the two quantities.

Link the number of particles n to the size of the interface l is made possible via a crucial approximation, self-labelled in this project as the flat-well approximation:

> The probability density of the Laughlin wave function can be approximately described by a flat distribution of density ρ_0 extending to radius r_0, where ρ_0 is defined as the plateau density around the origin and r_0 is defined as the maximum radius at which this density occurs.

This definition can be also be visualised to provide a clearer understanding. Using the density profile Fig.11b and extracting two of the individual curves, these are plotted in Fig.12 alongside a shaded region that represents the area under the curve according to the flat-well approximation.



(a) $n = 2$ particles (b) $n = 20$ particles

Figure 12: Looking at the 2-D density profile of the Laughlin wave function for individual values of n. Specifically, the approximation made in this project about the shape of the Laughlin probability density is represented by the shaded area.

Looking at Fig.12a, a first reaction might be to suggest that this approximation is in fact a

highly inaccurate representation of the true area under the curve. Nonetheless, by considering Fig.12b it is justified to suggest that as the number of particles increases, so to does the accuracy of the approximation. Indeed if we expect this trend to continue, for values where $n >> 1$ the approximation becomes almost an exact description of the truth. Ultimately, in any realistically sized condensed matter system (such as a quantum computer), this approximation behaves exactly as desired.

A physical insight into how the probability density of the Laughlin wave function evolves to become flatter and flatter can also be gleamed from Fig. 13. These plots were created from the 2-D density profile (Fig.11b) by first inverting and then rotating around the y-axis – an extrapolation which was safe to do because the Laughlin wave function is rotationally symmetric. The flat-well approximation can be interpreted in 3-D as asserting that the plots in Fig.13 are described by straight-walled wells, with radius r_0 and a flat bottom surface of depth of ${\rho_0}^5$. This alternative presentation of the density profiles allows a more physical interpretation of both the wave function and the purpose behind the flat-well approximation.

Surprisingly at the time of writing this report, research fails to yield any visualisations of the Laughlin wave functions similar to Fig.13 by other authors. As such, the author is excited to have produced a possibly unique perspective on the Laughlin states and it is hoped that the work done in creating them may benefit others who might also encounter the QHE in research of their own.

With this approximation we are now able to calculate the relationship between number of particles n and the boundary length l. The first step of this involves plotting the values ρ_0 against n, as is seen in Fig.14. A model function $y = \frac{1}{x}$ was fitted to the data using the scipy.optimize.curve_fit method, resulting in an equation of the form:

$$\rho_0 = \frac{0.323}{n^{1.015}} \approx \frac{1}{\pi n} \tag{13}$$

Where both estimates made here are within 1.5% of the true value, an uncertainty created by statistical fluctuations in generating MC data. When producing the density profiles ρ was normalised such that the integrated radial density was equal to 1, which is verified by the area under each curve in Fig.11a being equal to 1. Hence, we can construct the equation:

$$\pi r_0^2 \rho_0 = 1$$

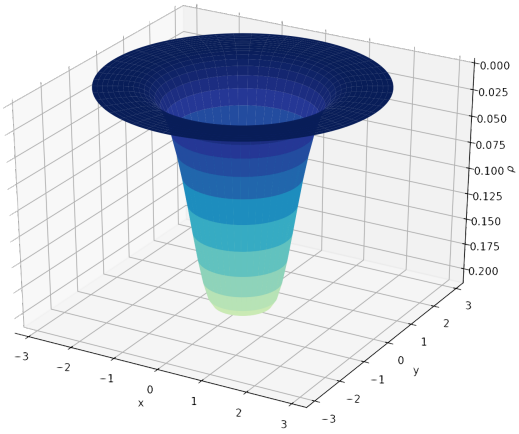
Substituting in Equation 13 we arrive at the relation:

$$r_0(n) = \sqrt{n}$$

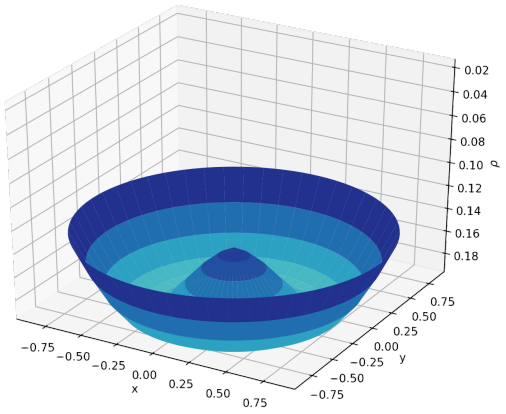
This becomes particularly significant when we recognise that the interface between subsections A and B is in fact the diameter of the density profile, exemplified in Fig.13f. Thus, $l = 2r_0$.

This is a huge breakthrough in the path to understanding the entanglement entropy because it is now testable if the Laughlin wave function obeys the area law. But if anyons are part fermion and part boson, which area law will they obey? An answer comes from the work of Zhang, Grover and Vishwanath[29], who state that the entanglement entropy of a topologically ordered phase in a two dimensional disk region obeys:

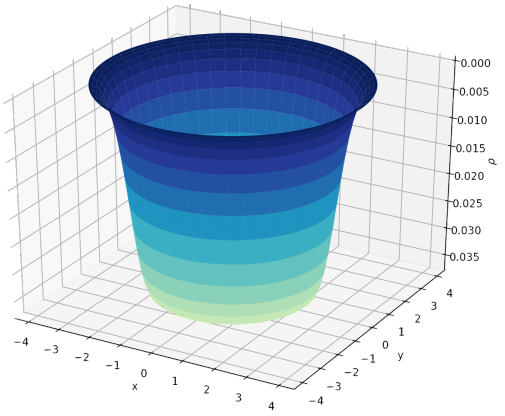
[5]From where the nomenclature flat-well approximation was coined.



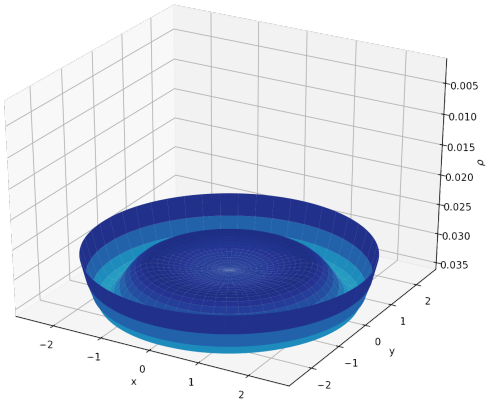
(a) n=2 particles



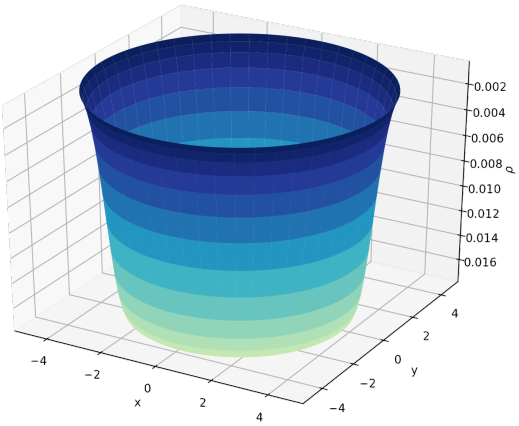
(b) Close-up of n=2 particles at the bottom of the well



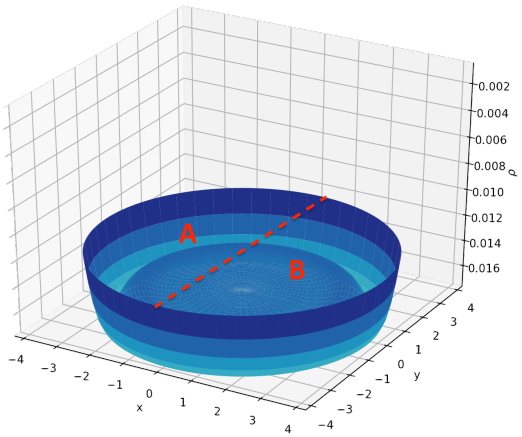
(c) n=10 particles



(d) Close-up of n=10 particles at the bottom of the well



(e) n=20 particles



(f) Close-up of n=20 particles at the bottom of the well

Figure 13: A 3-D visualisation of the 2-D particle density profile for the Laughlin wave function, using the data from Fig. 11b. The z axis has been inverted, such that there is a higher probability of observing a particle on a lower surface. This is analogous to how a classical particle would fall into the well, resulting in a higher probability of being observed at the bottom. Furthermore, there are additional annotations on (f), clarifying the definition of the subsystems used in calculating S_2.

$$S_2 = al_a - \gamma \tag{14}$$

Where a is a leading constant, l_a is the length of the interface and γ is known as the *topological entanglement entropy*. This equation is in fact a specific case of the more general original field theoretic results [30, 31], which state $\gamma = \log(D)$ where D is the quantum dimension of the phase. Subsequent works to analyse the quantum dimension of Laughlin states produces a more recognisable quantity $\gamma = \frac{1}{2}\log(m)$ [32, 33] where m is the index on the product term of the Laughlin wave function.

Taking Equation 14 and substituting $l = 2r_0 = 2\sqrt{n}$, a testable area law ansatz for our data emerges:

$$S2 = a\sqrt{n} - \gamma \tag{15}$$

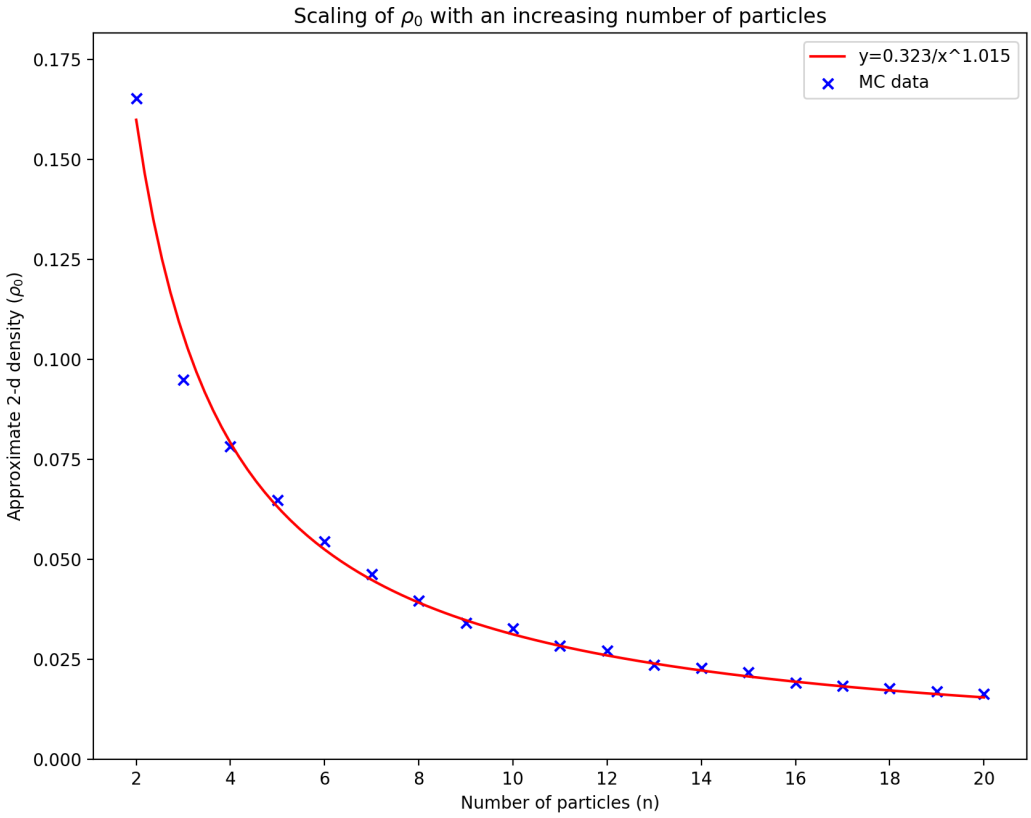


Figure 14: Graph showing the scaling of ρ_0 with the number of particles. Using the SciPy python package, a curve was fit of approximate equation $\rho_0 = \frac{1}{\pi n}$.

4.5 Results

As was achieved in the free-fermion case, a regime has now been constructed to directly test the area law by plotting S_2 vs n. Having adapted and improved the QMC algorithm, 15 runs were executed for $m = 1$ Laughlin states populated with a number of particles from $n = 2$ to $n = 16$. Each run comprised of 10 million iterations and deemed valid only if between 30-70% of the iterations were accepted. Furthermore, the standard error of the data points were produced via the same method as described in section 3.4. The results can be seen in Fig.15

A visual linear fit was applied using the scipy.stats.linregress with the results pushed to the graph legend. A further, more complex linear regression analysis was applied using the OLS linear regression model in Python's StatsModels package – which produced the uncertainties in each fitted variable.

Firstly, testing the goodness of the linear fit provides a quantitative measure of how well the results obey the fermionic area law. The OLS analysis reported a p-value of magnitude 10^{-22}, corresponding to the percentage probability of the null hypothesis being correct. In this case, the null hypothesis is that the results were arranged in a straight line by random probability without any underlying correlation. The extremely small p-value allows us to clearly reject such a hypothesis, demonstrating that the states being studied behave in strict compliance with the standard area law.

Verifying whether these results are representing the Laughlin state however requires analysis of the topological constant. This can be measured from the intercept of the graph, which was calculated via the OLS analysis as $\gamma = 0.1013 \pm 0.015$. Unfortunately, this is not in agreement with the theoretical result which for the m=1 state would be $\gamma = \frac{1}{2}\log(1) = 0$. The obvious candidate for where this discrepancy may have arisen is the flat-well approximation in section 4.4. This approximation proved to be least accurate for low values of n, therefore for further analysis Table 2 explores the effect of repeating the calculation, but discarding the data points below $n = n_{min}$ in linear fitting.

n_{min}	γ	$\Delta\gamma$
2	0.1013	0.015
3	0.0828	0.016
4	0.1034	0.016
5	0.0878	0.018

Table 2: Table showing the value and standard error of the topological entanglement entropy γ. A linear fit is applied as in Fig.15 to data points between n_{min} and $n = 16$.

Due to the limitations of available resources in this project, S_2 is only calculated up to a system size of n=16 particles. As such, extending Table 2 any further than $n_{min} = 5$ reduces the population of valid data points left to below 10. At this limit $\Delta\gamma$ starts to become significantly larger, making meaningful scrutiny of the results difficult. Whilst Table 2 does not provide a consistent trend, one could speculate that applying the algorithm to calculations starting at $n_{min} = 20$ would provide definite insight to whether γ is converging towards 0 or not. Regardless, the n_{min}=5 result strikes a balance between where the flat-well approximation is most accurate and where the uncertainty in γ has not largely increased. Because of this, it is this value of $\gamma = 0.0878 \pm 0.018$ that is quoted as our best result.

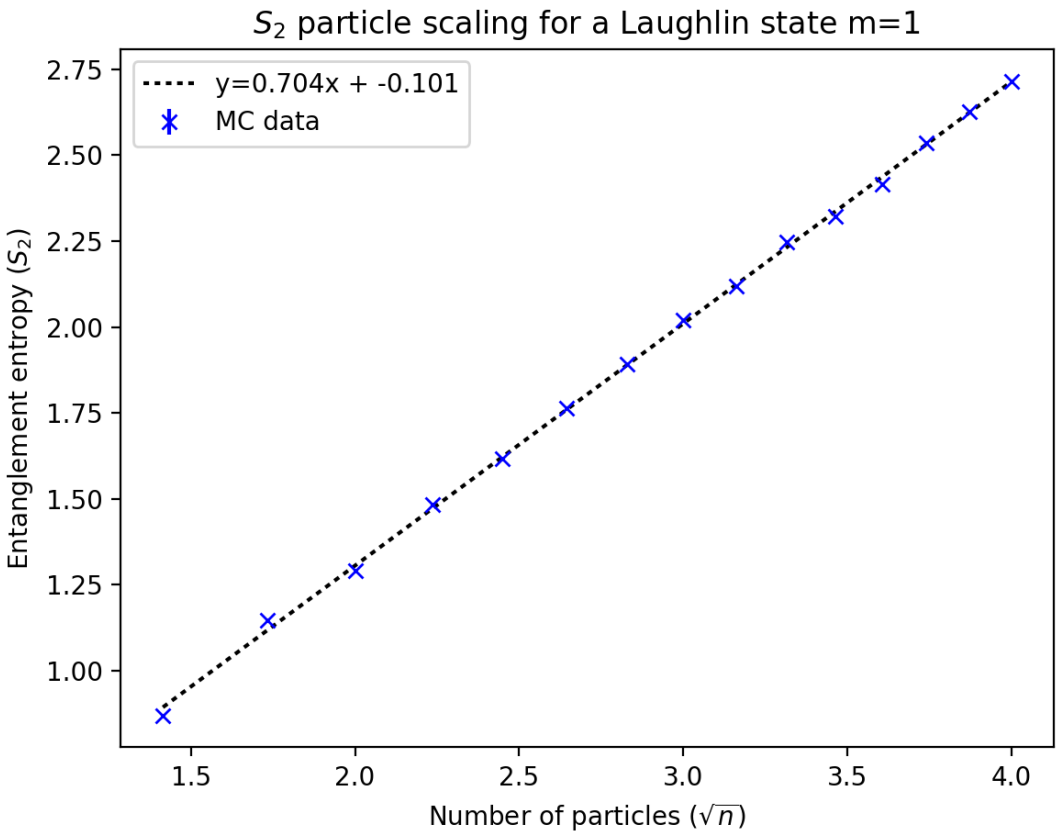


Figure 15: Graph showing the QMC calculated entanglement entropy S_2 of the Laughlin state m=1, versus the root of the number of particles in the system. The goodness of the linear fit corresponds to agreement with the fermionic area law, whilst the value of the intercept is a measure of the topological entanglement entropy. Note that the errors generated via the Monte Carlo method are present, but too small to be seen on this plot.

5 Conclusion

The original goal of this project was to build a Markov chain Monte Carlo algorithm able to calculate the entanglement entropy S_2 of the n-particle free-fermion wave function.

By developing understanding in the background theory of entanglement entropy, we were able to solve the mathematics behind various S_2 scaling behaviours in the free-fermion case. A short computer program was then written to use these algebraic results and produce graphs that corroborated the well established theory of the area law.

After this, research was conducted into the specific mechanics behind the Monte Carlo and Metropolis statistical methods. These were written into code, alongside an original implementation of the complex calculation scheme required to produce the value of S_2. Through superimposing the generated data onto the analytic results, we were able to graphically verify the validity of the constructed algorithm.

After fulfilling the initial goal, the project was extended to investigate the quantum Hall effect – chosen due to its non-traditional and highly exotic behaviour. Furthermore, the QHE is of particular interest due to the recent breakthroughs in understanding its non-trivial topology – for which the nobel prize in physics was awarded this year. An exploration into the science behind the quantum Hall effect enabled us to adapt the QMC algorithm to simulate particles represented by the Laughlin wave function. Through the flat-well approximation a regime was constructed by which our results could be compared to the topological entanglement entropy area law as described by recent publications. From this the topological constant was calculated to be $\gamma = 0.0878 \pm 0.018$, a significant deviation from the theoretical result $\gamma = 0$. We conclude that this is most likely due to the inaccuracy of the flat-well approximation for low numbers of particles – but remain cautious as to possible other flaws in our method.

At the time of writing, this project is a work in progress that is hoped to be continued beyond the scope of this thesis. Therefore, there are two main short term goals for which the results are hotly anticipated:

1. Repeat the QMC analysis of the Laughlin wave function for particle numbers $n > 20$. Hopefully this will confirm where the problem lies with the current inconsistency between our calculated value and the theory.
2. Produce similar results across several of the Laughlin states, such as $m = 3, 5, 7$. Calculating results in agreement with the theory for all of these states would substantiate the validity and effectiveness of our algorithm to generally solving S_2 for any wave function in the future.

Looking beyond the near future, a compelling long term goal for this project would be to study the quantum Hall effect as described by more advanced wave functions than studied here. As good as the Laughlin description may be, it is restricted by the condition that the filling fraction must be of the form $\nu = \frac{1}{m}$ where m is an odd integer. This does not explain certain QHE states that have been measured such as $\nu = \frac{2}{5}, \frac{3}{7}, \frac{5}{7}$... and so is lacking in our quest to understand the QHE on the most general level. Developments in the field have produced states able to explain these filling fractions, such as the Moore Read wave function [27]. For this reason, we would be very keen to perform a similar analysis using the QMC algorithm on the Moore Read states.

As a long term goal, it is recognised that pushing this project to the forefront of breakthrough scientific research would require incorporating a much deeper understanding of topology. The recent explosion of topology into the broader field of condensed matter is exemplified by its

ability to explain complex phenomenon such as the fractional QHE in a complete manner. In doing so, it would be immensely exciting to be able to apply our findings on the effect of topological order in entanglement entropy to the emerging prospect of topological quantum computing. This new and encouraging theory for a working quantum computer is fundamentally based on the topological description of anyons, and is therefore very acutely relevant to the interests of the author.

References

[1] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, 47(10):777–780, 1935.

[2] E. Schrodinger. Die gegenwartige situation in der quantenmechanik. *Die Naturwissenschaften*, 23(49):823–828, 1935.

[3] Brian Swingle. Entanglement entropy and the fermi surface. *Physical Review Letters*, 105(5), 2010.

[4] Brian Swingle. Conformal field theory approach to fermi liquids and other highly entangled states. *Phys. Rev. B*, 86(3), 2012.

[5] Andrea Coser, Luca Tagliacozzo, and Erik Tonni. On rnyi entropies of disjoint intervals in conformal field theory. *Journal of Statistical Mechanics: Theory and Experiment*, 2014(1):P01008, 2014.

[6] F Rajabpour, M AGliozzi. Entanglement entropy of two disjoint intervals from fusion algebra of twist fields. *Journal of Statistical Mechanics: Theory and Experiment*, 2012(02):P02016, 2012.

[7] Rex Lundgren, Jonathan Blair, Martin Greiter, Andreas Luchli, Gregory A. Fiete, and Ronny Thomale. Momentum-space entanglement spectrum of bosons and fermions with interactions. *Phys. Rev. Lett.*, 113(25), 2014.

[8] Vijay Balasubramanian, Michael B. McDermott, and Mark Van Raamsdonk. Momentum-space entanglement and renormalization in quantum field theory. *Physical Review D*, 86(4), 2012.

[9] Masudul Haque, Oleksandr Zozulya, and Kareljan Schoutens. Entanglement entropy in fermionic laughlin states. *Phys. Rev. Lett.*, 98(6), 2007.

[10] J. Eisert, M. Cramer, and M. B. Plenio. Colloquium : Area laws for the entanglement entropy. *Reviews of Modern Physics*, 82(1):277–306, 2010.

[11] Karol Zyczkowski. Renyi extrapolation of shannon entropy. *Open Systems & Information Dynamics (OSID)*, 10(03):297–310, 2003.

[12] Luca Bombelli, Rabinder K. Koul, Joohan Lee, and Rafael D. Sorkin. Quantum source of entropy for black holes. *Physical Review D*, 34(2):373–383, 1986.

[13] Nicolas Laflorencie. Quantum entanglement in condensed matter systems. *Physics Reports*, 646:1–59, 2016.

[14] P. Calabrese, M. Mintchev, and E. Vicari. Entanglement entropies in free-fermion gases for arbitrary dimension. *EPL (Europhysics Letters)*, 97(2):20009, 2012.

[15] Israel Gioev, DimitriKlich. Entanglement entropy of fermions in any dimension and the widom conjecture. *Physical Review Letters*, 96(10), 2006.

[16] Mohammad Pouranvari, Yuhui Zhang, and Kun Yang. Entanglement area law in disordered free fermion anderson model in one, two, and three dimensions. *Advances in Condensed Matter Physics*, 2015:1–4, 2015.

[17] Andrew J. Coleman, PiersSchofield. Quantum criticality. *Nature*, 433(7023):226–229, 2005.

[18] Junping Shao, Eun-Ah Kim, F. D. M. Haldane, and Edward H. Rezayi. Entanglement entropy of the ? = 1 / 2 composite fermion non-fermi liquid state. *Phys. Rev. Lett.*, 114(20), 2015.

[19] Pasquale Calabrese, Mihail Mintchev, and Ettore Vicari. Entanglement entropy of one-dimensional gases. *Physical Review Letters*, 107(2), 2011.

[20] Matthew B. Hastings, Ivn Gonzlez, Ann B. Kallin, and Roger G. Melko. Measuring renyi entanglement entropy in quantum montecarlo simulations. *Phys. Rev. Lett.*, 104(15), 2010.

[21] Yi Zhang, Tarun Grover, and Ashvin Vishwanath. Entanglement entropy of critical spin liquids. *Phys. Rev. Lett.*, 107(6), 2011.

[22] G. O. Roberts, A. Gelman, and W. R. Gilks. Weak convergence and optimal scaling of random walk metropolis algorithms. *The Annals of Applied Probability*, 7(1):110–120, 1997.

[23] Mylne Bdard. Optimal acceptance rates for metropolis algorithms: Moving beyond 0.234. *Stochastic Processes and their Applications*, 118(12):2198–2222, 2008.

[24] F. D. M. Haldane. Model for a quantum hall effect without landau levels: Condensed-matter realization of the "parity anomaly". *Physical Review Letters*, 61(18):2015–2018, 1988.

[25] D. J. Thouless, M. Kohmoto, M. P. Nightingale, and M. den Nijs. Quantized hall conductance in a two-dimensional periodic potential. *Physical Review Letters*, 49(6):405–408, 1982.

[26] Qian Niu, D. J. Thouless, and Yong-Shi Wu. Quantized hall conductance as a topological invariant. *Physical Review B*, 31(6):3372–3377, 1985.

[27] David Tong. Lectures on the quantum hall effect (arxiv:1606.06687, 2016.

[28] O. S. Zozulya, M. Haque, K. Schoutens, and E. H. Rezayi. Bipartite entanglement entropy in fractional quantum hall states. *Physical Review B*, 76(12), 2007.

[29] Yi Zhang, Tarun Grover, and Ashvin Vishwanath. Topological entanglement entropy ofz2spin liquids and lattice laughlin states. *Physical Review B*, 84(7), 2011.

[30] Alexei Kitaev and John Preskill. Topological entanglement entropy. *Physical Review Letters*, 96(11), 2006.

[31] Michael Levin and Xiao-Gang Wen. Detecting topological order in a ground state wave function. *Physical Review Letters*, 96(11), 2006.

[32] Masudul Haque, Oleksandr Zozulya, and Kareljan Schoutens. Entanglement entropy in fermionic laughlin states. *Physical Review Letters*, 98(6), 2007.

[33] B. A. Friedman and G. C. Levine. Topological entropy of realistic quantum hall wave functions. *Physical Review B*, 78(3), 2008.

[34] T. Hansson, D. Haviland, and G. Ingelman. *The Nobel Prize in Physics 2016: Popular Science Background*. The Royal Swedish Academy Of Sciences, 2016.

Appendices

A Topology

The field of topology concerns itself with properties of objects that remain unchanged when they are stretched, deformed or twisted – but not torn. Understanding what sort of physical property could obey such a condition is by no means immediately obvious. However, imagine starting with a sphere made of some soft flexible material and pressing your thumb into the top, such that it leaves an imprint. Through a relatively simple deformation we were able to create a different object – a bowl. In this way that a sphere and a bowl are topologically equivalent, so too are other everyday objects such as a doughnut and a mug.

However, some objects can never be made into one another by continuous deformations. Looking back at the previous example, it is clear that there is no way of turning a doughnut into a sphere without at some point tearing it. Therefore, it is apparent that one property used to define topological equality is the number of holes each object possesses. Fig.16 is an insightful graphic describing the definitions discussed above, which was created by the Royal Swedish Academy of Sciences in conjunction with the 2016 Nobel prize announcement [34]:

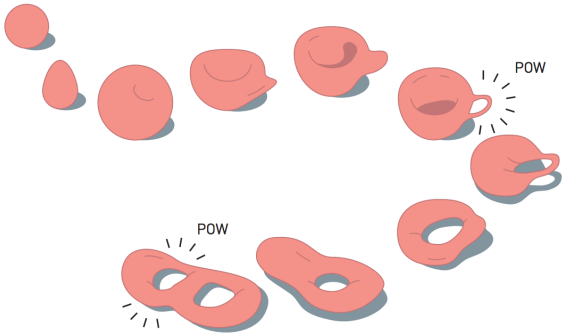


Figure 16: Graphic demonstrating the definition of key concepts in topology. Objects with the same number of holes can be continuously deformed into one another, however in order to move between different topological orders one must tear the material at some point.

The concept of topology offers a surprising alternate description for how conductance is quantised in the integer quantum Hall effect. Through topological arguments as followed in the freely available lecture course by David Tong [27], it is possible to show that the quantised conductance seen in Equation.11 can be independently derived in the form:

$$\sigma_{xy} = -\frac{e^2}{2\pi\hbar}C$$

where C is known as the Chern number and characterises the topological order of the system. This supplements the rather elegant qualitative observation, that the unbelievable precision of

ν in the integer QHE is because the number of holes in a system can only take exact integer values.

A.1 Topology as Applied to Anyons

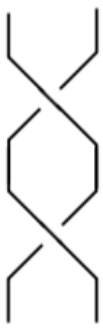
With this in mind, topology also offers an explanation to the existence of anyons in the fractional QHE. Firstly, recall that particles are defined as either fermions or bosons by the behaviour of their wave function under exchange. If we have two identical particles described by a wave function $\psi(\mathbf{r_1}, \mathbf{r_2})$, swapping them (rotating the system by 180 degrees) introduces a phase:

$$\psi(\mathbf{r_2}, \mathbf{r_1}) = e^{i\pi\alpha}\psi(\mathbf{r_1}, \mathbf{r_2})$$

Fermions are defined by antisymmetric wave functions such that $\psi(\mathbf{r_1}, \mathbf{r_2}) = -\psi(\mathbf{r_1}, \mathbf{r_2})$, which is achieved by setting $\alpha = 1$. Conversely we recover the definition of bosons which have symmetric wave functions by setting $\alpha = 0$. When swapped a second time, these are the only possible allowed values of α that allow us to recover the original wave function $\psi(\mathbf{r_1}, \mathbf{r_2})$, which we expect given the particles are back in their exact original position.

However, rather alarmingly, it is not always safe to assume that rotating two particles 360 degrees will leave them unchanged from the original state. Whilst this is true in 3 dimensions and higher, it is not for particles existing in 2 dimensions. Instead, the history of the particles end up wrapped around one another in a topological concept known as braiding. This results in fundamentally different outcomes depending on whether the particles are swapped clockwise or anti-clockwise, as shown in Fig.17:

(a) Clockwise



(b) Anti-clockwise

Figure 17: Diagram showing the swapping of two identical particles in two dimensions. Unlike in 3-D, the spatial history of the particles overlap such that there becomes a difference in whether they are rotated clockwise or anticlockwise.

Because the particles can never be described as returning to their original state, we are no longer restricted by the condition that $\alpha = 0, 1$. This opens up the possibility of particles existing outside of the binary definition of either fermions or bosons! In this way, the existence of anyons is justified as at least a theoretical concept.

B Code

B.1 Point Class

This was the custom Point class built to represent the position and spatial properties of a particle in two dimensions.

```
1  //
2  // point.cpp
3  // Kf_Calculator
4  //
5  // Created by Benjamin Jaderberg on 12/10/2016.
6  // Copyright  2016 BenJad. All rights reserved.
7  //
8
9  #include "Point.hpp"
10 #include <iostream>
11 #include <math.h>
12 #include <stdio.h>
13
14
15 //Constructor
16 Point::Point(double x, double y) {
17     xval = x;
18     yval = y;
19 }
20
21 //Getter method
22 double Point::x() { return xval; }
23 double Point::y() { return yval; }
24
25 // Distance to another point. Pythagorean thm.
26 double Point::dist(Point other) {
27     double xd = xval - other.xval;
28     double yd = yval - other.yval;
29     return sqrt(xd*xd + yd*yd);
30 }
31
32 //Length of a vector point
33 double Point::length() {
34     return sqrt(xval*xval + yval*yval);
35 }
36
37 // Add or subtract two points.
38 Point Point::add(Point b)
39 {
40     return Point(xval + b.xval, yval + b.yval);
41 }
42 Point Point::sub(Point b)
43 {
44     return Point(xval - b.xval, yval - b.yval);
45 }
46
47 // Move the existing point.
48 void Point::move(double a, double b)
```

```
{
    xval += a;
    yval += b;
}

// Print the point on the stream. The class ostream is a base class
// for output streams of various types.
void Point::print()
{
    std::cout << "(" << xval << "," << yval << ")";
}
```

B.2 Free-Fermion Analytic Calculation

B.2.1 Helper Class of Useful Methods

```
//
// methods.cpp
// Kf_Calculator
//
// Created by Benjamin Jaderberg on 28/10/2016.
// Copyright  2016 BenJad. All rights reserved.
//

#include "methods.hpp"
#include "Point.hpp"
#include <vector>
#include <fstream>
#include <iostream>

using namespace std;

//Method to create a grid of a given point spacing and size
vector<Point> createGrid(int spacing, int size) {
    vector<Point> grid;
    for (int i=-size; i<=size; i=i+spacing) {
        for (int j=-size; j<=size; j=j+spacing) {
            Point newPoint = Point(i,j);
            grid.push_back(newPoint);
        }
    }
    return grid;
}

//Method to find the points within a radius r
vector<Point> findPointsWithin(double radius, vector<Point> grid) {
    vector<Point> pointsWithinRange;
    for (Point p : grid) {
        if (p.length() < radius) {
            pointsWithinRange.push_back(p);
        }
    }
    return pointsWithinRange;
}
```

```
39
40 //Method to find number of points within a radius r
41 int findNumPointsWithin(double radius, vector<Point> grid) {
42     int counter = 0;
43     for (Point p : grid) {
44         if (p.length() < radius) {
45             counter ++;
46         }
47     }
48     return counter;
49 }
50
51
52 //Method to iterate findNumPoints
53 vector<Point> iterateNumPoints(double maxR, int steps, vector<Point> grid) {
54
55     double increment = maxR / steps;
56     double r = 0.0;
57     vector<Point> list;
58
59     while (r<maxR) {
60
61         double numPoints = findNumPointsWithin(r, grid);
62         Point onePoint = Point(r,numPoints);
63         list.push_back(onePoint);
64         r = r + increment;
65     }
66     return list;
67 }
68
69 //Method to find the radius of the smallest shell that fills N points
70 double findShellRadius(int n, vector<Point> grid) {
71
72     double radius = 0.0;
73     //Find the number of points in many shells over a range of radii
74     vector<Point> shellNumbers = iterateNumPoints(10, 500, grid);
75
76     for (Point p : shellNumbers) {
77         if (p.y() == n) {
78             radius = p.x();
79             break;
80         }
81     }
82
83     return radius;
84 }
85
86 //Method to write a vector of points to comma seperated file
87 void writeToFile(vector<Point> list, string name) {
88     ofstream myfile("/Users/benjaminjaderberg/Desktop/4th_Year/MSci_Project/Section1/
        Kf_Calculator/" + name);
89     if (myfile.is_open()) {
90         for (Point p : list) {
91             myfile << p.x() << "," << p.y() << "\n";
92         }
```

```
93          myfile.close();
94      }
95      else { cout << "Writing to file failed";}
96  }
```

B.2.2 Main Class

```
1   //
2   // main.cpp
3   // Kf_Calculator
4   //
5   // Created by Benjamin Jaderberg on 12/10/2016.
6   // Copyright  2016 BenJad. All rights reserved.
7   //
8
9   #include "Point.hpp"
10  #include "methods.hpp"
11
12  #include <cmath>
13  #include <complex>
14  #include <Eigen/Dense>
15  #include <Eigen/Eigenvalues>
16
17  using namespace std;
18  using namespace Eigen;
19
20  const complex<double> i(0.0,1.0);
21  double L = 1;
22  double l = 0.5;
23  double rho = 1.0;
24
25  //Method to compute the integral g(k) (see report)
26
27  double g (double k) {
28
29      double g_k = (1/(k*L)) * (2*sin((k*l)/2));
30
31      return g_k;
32  }
33
34  //Method to calculate the overlap value of two free fermionic wave functions using
        hand derived result of the integration
35  complex<double> calcOverlapElement(Point point_i, Point point_j) {
36
37      double deltaKx = (2*M_PI / L) * (point_i.x() - point_j.x());
38      double deltaKy = (2*M_PI / L) * (point_i.y() - point_j.y());
39
40      complex<double> xComponent, yComponent;
41
42      //If deltaX/Y is NON ZERO:
43      if (deltaKx != 0) {
44          xComponent = g(deltaKx);
45      } else {
46          xComponent = l/L;
47      }
```

```
48
49      if (deltaKy != 0) {
50          yComponent = g(deltaKy);
51      } else if (deltaKy == 0) {
52          yComponent = l/L;
53      }
54
55      complex<double> overlapElement = xComponent * yComponent;
56
57      return overlapElement;
58  }
59
60  //Method to create overlap matrix
61  MatrixXd createOverlapMatrix(vector<Point> pointsWithin) {
62
63      //Create an empty doubles matrix of size nxn
64      MatrixXd overlapMatrix;
65      overlapMatrix.resize(pointsWithin.size(), pointsWithin.size());
66
67      for (int j = 0; j < pointsWithin.size(); j++) {
68          for (int k=0; k< pointsWithin.size(); k++) {
69              complex<double> overlap = calcOverlapElement(pointsWithin[j], pointsWithin[
                    k]);
70
71              //In our case we know the matrix will be real
72              overlapMatrix(j, k) = overlap.real();
73          }
74      }
75      return overlapMatrix;
76  }
77
78  //Method to calculate the Renyi entanglement entropy S_alpha
79  double calcS_alpha(double alpha, VectorXd eigenvalues) {
80
81      double s_alpha = 0;
82
83      for (int i = 0; i<eigenvalues.size(); i++) {
84          double lambda = abs(eigenvalues[i]);
85          double f_alpha;
86
87          if (alpha==1.0) {
88              f_alpha = ((lambda-1)*log(1-lambda)) - lambda*log(lambda);
89          }
90          else {
91              f_alpha = (1/(1-alpha))*log((pow(lambda,alpha))+pow((1-lambda),alpha));
92          }
93
94          s_alpha = s_alpha + f_alpha;
95      }
96
97      return s_alpha;
98  }
99
100 //Method to calculate S_alpha over a range of z = l/L for fixed number
101
```

```
vector<Point> iterateS_alpha_l(double alpha, double steps, vector<Point> pointsWithin) {

    vector<Point> s_alphaPoints;
    double increment = L/steps;
    l=increment;

    while (l<L) {
        MatrixXd overlapMatrix = createOverlapMatrix(pointsWithin);
        SelfAdjointEigenSolver<MatrixXd> solver(overlapMatrix);
        VectorXd eigenvalues = solver.eigenvalues();

        double s_alpha = calcS_alpha(alpha, eigenvalues);
        Point onePoint = Point(l/L,s_alpha);

        s_alphaPoints.push_back(onePoint);

        l = l + increment;
    }

    return s_alphaPoints;
}

//Method to calcualte S_alpha over all possible shells in the grid for fixed z = l/L

vector<Point> iterateS_alpha_n(double alpha, double z, vector<Point> grid) {
    vector<Point> s_alphaPoints;

    //Create a list of all shell numbers
    vector<Point> nonDegenShells = iterateNumPoints(10, 500, grid);

    for (Point p1 : nonDegenShells) {
        int n = p1.y();
        double r = p1.x();
        bool duplicate = false;

        //Check that this shell has not already been added
        for (Point p2 : s_alphaPoints) {
            if (n == p2.x()) {
                duplicate = true;
            }
        }

        if (duplicate == false && r!=0) {
            //Adjust size of partitions for n - allowing constant density ro = 1
            L = sqrt(n/rho);
            l = z*L;

            vector<Point> pointsWithin = findPointsWithin(r, grid);

            MatrixXd overlapMatrix = createOverlapMatrix(pointsWithin);
            SelfAdjointEigenSolver<MatrixXd> solver(overlapMatrix);
            VectorXd eigenvalues = solver.eigenvalues();

            double s_alpha = abs(calcS_alpha(alpha, eigenvalues));
```

```
156             s_alphaPoints.push_back(Point(n, s_alpha));
157         }
158     }
159     return s_alphaPoints;
160 }
161
162 int main(int argc, const char * argv[]) {
163
164     //Create a grid
165     vector<Point> grid = createGrid(1, 10);
166     cout << "The␣grid␣has␣" << grid.size() << "␣points␣within␣it,␣ranging␣from␣";
167     grid[0].print();
168     cout << "␣to␣";
169     grid[grid.size()-1].print();*/
170
171
172     //TASK ONE - Calculate the number of points within a shell as a function of the
            radius of the shell
173
174     //Find the number of points as a function of r:
175     double maxR = 10.0;
176     int steps = 500;
177     vector<Point> list = iterateNumPoints(maxR, steps, grid);
178
179     //Write the graph data to "kf_data.txt"
180     //writeToFile(list, "kf_data.txt");
181
182     //TASK TWO - For a fixed n=37 particles, calculate S_alpha as a function of l/L
183
184     //For n particles in a non-degenerate shell, find the smallest radius that fills
            it
185     double r37 = findShellRadius(37, grid);
186
187     //Find all the momenta points within that radius
188     vector<Point> pointsWithin = findPointsWithin(r37, grid);
189
190     //Calculate S_2 over 100 l values
191     vector<Point> s_2Points = iterateS_alpha_l(2, 100, pointsWithin);
192     writeToFile(s_2Points, "s2_vs_l.txt");
193
194     //And for S_1, S_3, S_4
195     vector<Point> s_1Points = iterateS_alpha_l(1, 100, pointsWithin);
196     writeToFile(s_1Points, "s1_vs_l.txt");
197     vector<Point> s_3Points = iterateS_alpha_l(3, 100, pointsWithin);
198     writeToFile(s_3Points, "s3_vs_l.txt");
199     vector<Point> s_4Points = iterateS_alpha_l(4, 100, pointsWithin);
200     writeToFile(s_4Points, "s4_vs_l.txt");
201
202     //TASK THREE - For different numbers of particles, calculate S_2 as a function of
            l/L
203
204     int n1 = 5, n2 = 13, n3 = 25, n4 = 49, n5 = 81;
205     int nList [5] = {n1,n2,n3,n4,n5};
206
207     for (int i = 0; i< 5; i++) {
```

```
        double rN = findShellRadius(nList[i], grid);
        vector<Point> pointsWithin = findPointsWithin(rN, grid);
        writeToFile(iterateS_alpha_1(2, 100, pointsWithin), "s2_n" + to_string(nList[i
            ]) + ".txt");
    }

    //TASK FOUR - For fixed z, calcualte S_2 as a function of N

    double z1 = 0.1, z2 = 0.2, z3 = 0.4, z4 = 0.8;
    double zList [4] = {z1,z2,z3,z4};

    for (int i = 0 ; i<4; i++) {
        vector<Point> points = iterateS_alpha_n(2, zList[i], grid);
        writeToFile(points, "s2_vs_n_z" + to_string(i+1) + ".txt");
    }



}
```

B.3 QMC Algorithm for Free-Fermions

B.3.1 Methods

Same as B.2.1 but with the following additional method:

```
//Method to write MC generated data to file, a vector of Points and a vector of stdev
    doubles
void writeMCToFile(vector<Point> list, vector<double> stdev, string name) {
    ofstream myfile("/Users/benjaminjaderberg/Desktop/4th_Year/MSci_Project/Section2/"
        + name);
    if (myfile.is_open()) {
        for (int i=0; i<list.size(); i++) {
            Point p = list[i];
            myfile << p.x() << "," << p.y() << "," << stdev[i] << "\n";
        }
        myfile.close();
    }
    else { cout << "Writing␣to␣file␣failed";}
}
```

B.3.2 Main Class

```
//
// main.cpp
// Section2
//
// Created by Benjamin Jaderberg on 23/11/2016.
// Copyright  2016 BenJad. All rights reserved.
//

#include <iostream>

#include "Point.hpp"
```

```
12 #include "methods.hpp"
13
14 #include <cmath>
15 #include <complex>
16 #include <random>
17 #include <algorithm>
18 #include <Eigen/Dense>
19 //#include <Eigen/Eigenvalues>
20
21 using namespace std;
22 using namespace Eigen;
23
24 //Global constants
25 const complex<double> i(0.0,1.0);
26 double L = 1;
27 double l = 0.5;
28 double z = l/L;
29
30 //Global objects
31 random_device rd;
32 //Mersenne Twister algorithm, default seed
33 mt19937 gen(rd());
34 uniform_real_distribution<> dis(0,1); // Define our distribution as between 0 and 1
35
36 //Method to calculate the slater determinant of a system
37
38 complex<double> calcSlaterDet (vector<Point> kPoints, vector<Point> rPoints) {
39
40     //Initialise empty matrix of correct size
41     MatrixXcd matrix;
42     matrix.resize(kPoints.size(), kPoints.size());
43
44     //Fill the matrix with points exp(i*k_i*r_i)
45     for (int t=0; t<kPoints.size(); t++) {
46         for (int u=0; u<rPoints.size(); u++) {
47             Point k_i = kPoints[t];
48             Point r_i = rPoints[u];
49             complex<double> element = exp(i*(2*M_PI / L)*(k_i.dot(r_i)));
50
51             matrix(t,u) = element;
52
53         }
54     }
55     complex<double> slater_det = matrix.determinant();
56     return slater_det;
57 }
58
59 //Method to calculate the probability over two given wave function: p = |psi1|^2 * |
       psi2|^2
60
61 double calcProbability (complex<double> psi1, complex<double> psi2) {
62
63     return ((pow(psi1.real(),2)) + (pow(psi1.imag(),2))) * ((pow(psi2.real(),2)) + (
           pow(psi2.imag(),2)));
64 }
```

```
65
66 //Method using a Metropolis algorithm to iterate the two systems over random moves
67
68 vector<Point> runMetropolis (vector<Point> kPoints, vector<Point> rPoints1, vector<
       Point> rPoints2, double dr, int num_iterations) {
69
70     //Define some counters
71     int accepted = 0;
72     int rejected = 0;
73     int swapped = 0;
74
75     //Set up array to keep track of integral summations for a set number of Z values
76     int num_z = 9;
77     vector<double> z_values;
78     for (int b=0; b < num_z; b++) {
79         z_values.push_back((1.0/(num_z+1))*(b+1));
80     }
81     vector<complex<double>> sum_gx;
82     vector<complex<double>> sum_g2x;
83     for (int a=0; a<num_z; a++) {
84         sum_gx.push_back(0);
85         sum_g2x.push_back(0);
86     }
87
88     //Define current system
89     vector<Point> R1(rPoints1);
90     vector<Point> R2(rPoints2);
91
92     for (int i = 0; i<num_iterations; i++) {
93
94         //Calculate probability of current system
95         complex<double> psi1 = calcSlaterDet(kPoints, R1);
96         complex<double> psi2 = calcSlaterDet(kPoints, R2);
97         double p = calcProbability(psi1, psi2);
98
99         //Create empty test system
100        vector<Point> R3;
101        vector<Point> R4;
102
103        //Populate new system with original particles plus a random displacement of
               length dr
104        for (Point p1 : R1) {
105            double phi = (2*M_PI) * dis(gen);
106            double x = p1.x() + (dr * cos(phi));
107            double y = p1.y() + (dr * sin(phi));
108
109            //If a particle moves outside the box, instead make it reappear out the
                   opposite side
110            if (x > (L/2)) { x = x - L;}
111            if (x < (-L/2)) {x = x + L;}
112            if (y > (L/2)) { y = y - L;}
113            if (y < (-L/2)) { y = y + L;}
114
115            R3.push_back(Point(x,y));
116        }
```

```
117         for (Point p2 : R2) {
118             double phi = (2*M_PI) * dis(gen);
119             double x = p2.x() + (dr * cos(phi));
120             double y = p2.y() + (dr * sin(phi));
121 
122             if (x > (L/2)) { x = x - L;}
123             if (x < (-L/2)) {x = x + L;}
124             if (y > (L/2)) { y = y - L;}
125             if (y < (-L/2)) { y = y + L;}
126 
127             R4.push_back(Point(x,y));
128 
129         }
130 
131         //Calculate probability of new system
132         complex<double> psi3 = calcSlaterDet(kPoints, R3);
133         complex<double> psi4 = calcSlaterDet(kPoints, R4);
134         double p_new = calcProbability(psi3,psi4);
135 
136         //Accept in accordance to Hastings-Metropolis method
137         double lambda = min(p_new/p,1.0);
138         double alpha = dis(gen);
139 
140         //If accept the new configuration
141         if (alpha < lambda) {
142 
143             accepted += 1;
144             complex<double> g;
145             vector<int> iListR3;
146             vector<int> iListR4;
147 
148             //The new moved system becomes our current system
149             R1 = R3;
150             psi1 = psi3;
151             R2 = R4;
152             psi2 = psi4;
153             //We will keep using R3 and R4 as the system for our swap calculation
154 
155             //LOOP OVER L HERE
156             for (int k=0; k < num_z ; k++) {
157 
158                 //Assign size of l from specified values
159                 l = z_values[k]*L;
160 
161                 //Reset the swapped systems used for a previous l value
162                 R3 = R1; R4 = R2;
163 
164                 //Find the ID of particles within subsystem A for each box
165                 for (int j=0; j<R3.size(); j++) {
166                     if (abs(R3[j].x()) < l/2 && abs(R3[j].y()) < l/2) {
167                         iListR3.push_back(j);
168                     }
169                     if (abs(R4[j].x()) < l/2 && abs(R4[j].y()) < l/2) {
170                         iListR4.push_back(j);
171                     }
```

```
            }
            //If the number of particles in R3_A = R4_A we can perform the swap
            if (iListR3.size() == iListR4.size()) {
                swapped += 1;
                for (int d=0; d<iListR3.size(); d++) {

                    //Find the index integer of the particles we are swapping
                    int id_1 = iListR3[d];
                    int id_2 = iListR4[d];

                    //Swap the particles by copying each others coordinates
                    //At this point R2 is still an exact copy of R4. Copying from R2
                         avoids creating a temporary swap variable.
                    R3[id_1].copy(R2[id_2]);
                    R4[id_2].copy(R1[id_1]);
                }
                //Recalculate the wavefunctions of R3 and R4 (after the swap)

                psi3 = calcSlaterDet(kPoints, R3);
                psi4 = calcSlaterDet(kPoints, R4);
                g = (psi3 * psi4)/(psi1 * psi2);

            }
            //If we didnt perform the swap, dont add to the integral (dirac delta
                factor)
            else {
                g = 0;
            }
            sum_gx[k] += g;
            sum_g2x[k] += pow(g,2);

            iListR3.clear();
            iListR4.clear();
        }
    }
    //Else reject the new configuration
    else {
        rejected +=1;
    }

}
//Calculate the error of each S2 point
vector<double> normIntegral;
for (int b=0; b<sum_gx.size(); b++) {
    normIntegral.push_back((sum_gx[b].real() / accepted));
}
vector<double> normSquaredIntegral;
for (int i=0; i<sum_g2x.size(); i++) {
    normSquaredIntegral.push_back(sum_g2x[i].real() / accepted);
}

vector<double> stdevS2;
for (int i=0; i<sum_gx.size(); i++) {
    stdevS2.push_back((pow(normSquaredIntegral[i] - pow(normIntegral[i],2),0.5))/(
        pow(accepted,0.5)*normIntegral[i]));
```

```
224     }
225     vector<double> s2;
226 
227     //Print properties of run
228     cout << endl << "Number␣of␣accepted␣moves:␣" << accepted;
229     cout << endl << "Number␣of␣rejected␣moves:␣" << rejected;
230     cout << endl << "Acceptance␣rate:␣" << (double(accepted)/(accepted + rejected))
            *100 << "%";
231     cout << endl << "Number␣of␣swaps:␣" << swapped;
232 
233     //Print normalised integral and calculate S2 for each one
234     cout << endl << "Normalised␣integrals:␣";
235     for (double d : normIntegral) {
236         cout << d << "␣";
237         s2.push_back(-log(d));
238     }
239 
240     //Print S2 with the standard deviation
241     cout << endl << "S2␣values:␣";
242     for (int i = 0; i<s2.size(); i++) {
243         cout << s2[i] << "+/-" << stdevS2[i] << endl;
244     }
245 
246     //Create the graph Points that will be plotted (Z vs S2)
247     vector<Point> s2Points;
248     for (int i=0; i<s2.size(); i++) {
249         s2Points.push_back(Point(z_values[i], s2[i]));
250     }
251 
252     //Write results to file for plotting
253     string file_name = "MC_n" + to_string(rPoints1.size()) + "_" + to_string(
            num_iterations/1000) + "k.txt";
254     writeMCToFile(s2Points, stdevS2, file_name);
255     return s2Points;
256 
257 }
258 
259 int main(int argc, const char * argv[]) {
260 
261     //Initialise two empty systems
262     vector<Point> R1;
263     vector<Point> R2;
264     double r = 4.05; //Radius of k-space defined
265 
266     //Find the k-space coordinates that will be occupied
267     vector<Point> kGrid = createGrid(1, r);
268     vector<Point> kPoints = findPointsWithin(r, kGrid);
269 
270     //Initialise the subsystems with random points
271     for (int i=0; i<kPoints.size(); i++) {
272         double rand_x1 = dis(gen)-0.5; //Random numbers between -0.5 and 0.5
273         double rand_y1 = dis(gen)-0.5;
274         double rand_x2 = dis(gen)-0.5;
275         double rand_y2 = dis(gen)-0.5;
276 
```

```
277         Point r1 = Point(rand_x1 * L,rand_y1 * L); //Vector r_i has random components
                between -L/2 and L/2
278         Point r2 = Point(rand_x2 * L,rand_y2 * L);
279         R1.push_back(r1);
280         R2.push_back(r2);
281     }
282     //Run the Monte Carlo Metropolis algorithm for a certin number of iterations
283     vector<Point> S2 = runMetropolis(kPoints, R1, R2, 0.005, 1000000);
284 
285 }
```

B.4 QMC Algorithm for Laughlin States

B.4.1 Methods

```
1  //
2  // methods.cpp
3  // Section3
4  //
5  // Created by Benjamin Jaderberg on 17/01/2017.
6  // Copyright  2017 BenJad. All rights reserved.
7  //
8  
9  #include "methods.hpp"
10 
11 //Method to write MC generated data to file, a vector of Points and a vector of stdev
       doubles
12 void writeMCToFile(vector<vector<double>> list, string name) {
13     ofstream myfile("/Users/benjaminjaderberg/Desktop/4th_Year/MSci_Project/Section3/"
           + name);
14     if (myfile.is_open()) {
15         for (int i=0; i<list.size(); i++) {
16             myfile << (list[i])[0] << "," << (list[i])[1] << "," << (list[i])[2] << "\n
                   ";
17         }
18         myfile.close();
19     }
20     else { cout << "Writing␣to␣file␣failed";}
21 }
22 
23 //Method to write a vector of points to comma seperated file
24 void writeToFile(vector<Point> list, string name) {
25     ofstream myfile("/Users/benjaminjaderberg/Desktop/4th_Year/MSci_Project/Section3/"
           + name);
26     if (myfile.is_open()) {
27         for (Point p : list) {
28             myfile << p.x() << "," << p.y() << "\n";
29         }
30         myfile.close();
31     }
32     else { cout << "Writing␣to␣file␣failed";}
33 }
34 
35 
36 //Method to calculate the probability over two given wave function: p = |psi1|^2 * |
```

```
    psi2|^2
double calcJointProb (complex<double> psi1, complex<double> psi2) {

    return norm(psi1) * norm(psi2);
}

//Method to create a Laughlin wavefunction based off input coordinates in 2-d space
complex<double> createWaveFunction(vector<Point> points) {

    double scale_factor = 1.9;
    int m=3;

    complex<double> Psi;
    vector<complex<double>> zPoints;

    for (Point p : points) {
        zPoints.push_back(complex<double>(p.x(),p.y()));
    }

    //Initialise summation variables
    complex<double> product = complex<double>(1.0,0.0);
    double sum_square = 0;

    //For every point in the system
    for (int i=0; i<zPoints.size(); i++) {

        complex<double> z_i = zPoints[i];

        for (int j=0; j<zPoints.size(); j++) {

            complex<double> z_j = zPoints[j];
            complex<double> product_term = pow((z_i - z_j),m);
            //cout << endl << product_term;

            if(product_term == complex<double>(0.0,0.0)) {
            }
            else {
                product = product * (product_term/scale_factor);
            }

        }

        //Add |Z|^2 to the exponent sum
        sum_square += norm(z_i);
    }

    Psi = product * exp(-sum_square);

    return Psi;
}

//Method to calculate only one term of the Laughlin wave function
complex<double> createWaveFunctionTerm(vector<Point> points, int option) {

    double scale_factor = 1.9;
```

```
    int m=3;

    complex<double> Psi;
    vector<complex<double>> zPoints;

    for (Point p : points) {
        zPoints.push_back(complex<double>(p.x(),p.y()));
    }

    //Initialise summation variables
    complex<double> product = complex<double>(1.0,0.0);
    double sum_square = 0;

    //For every point in the system
    for (int i=0; i<zPoints.size(); i++) {

        complex<double> z_i = zPoints[i];

        for (int j=0; j<zPoints.size(); j++) {

            complex<double> z_j = zPoints[j];
            complex<double> product_term = pow((z_i - z_j),m);

            if(product_term == complex<double>(0.0,0.0)) {
            }
            else {
                product = product * (product_term/scale_factor);
            }

        }

        //Add |Z|^2 to the exponent sum
        sum_square += norm(z_i);
    }

    complex<double> exp_term = exp(-sum_square);

    //Return the desired part of the wave function
    if (option) {
        return exp_term;
    }
    else {
        return product;
    }
}
```

B.4.2 Main Class

```
//
// main.cpp
// Section3
//
// Created by Benjamin Jaderberg on 17/01/2017.
// Copyright  2017 BenJad. All rights reserved.
//
```

```
 8
 9  #include <iostream>
10
11  #include "Point.hpp"
12  #include "methods.hpp"
13
14  #include <cmath>
15  #include <complex>
16  #include <random>
17  #include <algorithm>
18  #include <Eigen/Dense>
19
20  using namespace std;
21  using namespace Eigen;
22
23  //Global constants
24  //const complex<double> i(0.0,1.0);
25  double L = 10;
26
27  //Global objects
28  random_device rd;
29  //Mersenne Twister algorithm, default seed
30  mt19937 gen(rd());
31  uniform_real_distribution<> dis(0,1); // Define our distribution as between 0 and 1
32
33  //Method to initialise a system with N particles randomly distributed within 3-sigma
        radius
34  vector<Point> initialiseSystem (int N) {
35
36      vector<Point> R;
37
38      for (int i=0; i<N; i++) {
39
40          //Populate R1 with particles randomly placed within radius of 3sqrt(N)-sigma
                from origin
41          double sigma = (1/sqrt(2))*sqrt(N);
42
43          double phi1 = (2*M_PI) * dis(gen); //Random angle in polar coordinates
44          double radius1 = (dis(gen) * 4 * sigma); //Random radius between 0 -> 3-sigma
45
46          double x1 = radius1 * cos(phi1);
47          double y1 = radius1 * sin(phi1);
48          R.push_back(Point(x1,y1));
49      }
50      return R;
51  }
52
53  //Method to "burn in" a random system to more accurately represent the true
        distribution
54  //Runs same Metropolis algorithm as runMetropolis but without any calculations or
        swaps
55  void runBurnIn (vector<Point> &R1, vector<Point> &R2, double dr, int num_iterations) {
56
57      for (int i = 0; i<num_iterations; i++) {
58
```

```
59          //Calculate probability of current system
60          complex<double> psi1 = createWaveFunction(R1);
61          complex<double> psi2 = createWaveFunction(R2);
62          double p = calcJointProb(psi1, psi2);
63
64          //Create empty test system
65          vector<Point> R3;
66          vector<Point> R4;
67
68          //Populate new system R3 with original particles from R1, plus a random
                displacement of length dr
69          for (Point p1 : R1) {
70              double phi = (2*M_PI) * dis(gen);
71              double x = p1.x() + (dr * cos(phi));
72              double y = p1.y() + (dr * sin(phi));
73
74              //If a particle moves outside the box, instead make it reappear out the
                    opposite side
75              if (x > (L/2)) { x = x - L;}
76              if (x < (-L/2)) {x = x + L;}
77              if (y > (L/2)) { y = y - L;}
78              if (y < (-L/2)) { y = y + L;}
79
80              R3.push_back(Point(x,y));
81          }
82
83          //Repeat for R4/R2
84          for (Point p2 : R2) {
85              double phi = (2*M_PI) * dis(gen);
86              double x = p2.x() + (dr * cos(phi));
87              double y = p2.y() + (dr * sin(phi));
88
89              if (x > (L/2)) { x = x - L;}
90              if (x < (-L/2)) {x = x + L;}
91              if (y > (L/2)) { y = y - L;}
92              if (y < (-L/2)) { y = y + L;}
93
94              R4.push_back(Point(x,y));
95
96          }
97
98          //Calculate probability of new system
99          complex<double> psi3 = createWaveFunction(R3);
100         complex<double> psi4 = createWaveFunction(R4);
101         double p_new = calcJointProb(psi3,psi4);
102
103         //Accept in accordance to Hastings-Metropolis method
104         double lambda = min(p_new/p,1.0);
105         double alpha = dis(gen);
106
107         //If accept the new configuration
108         if (alpha < lambda) {
109
110             R1 = R3;
111             psi1 = psi3;
```

```
112             R2 = R4;
113             psi2 = psi4;
114         }
115 
116     }
117 }
118 
119 //Method to find the density profile of N-particles
120 
121 vector<Point> densityProfile (vector<Point> R1, double dr, int num_iterations, double
        width) {
122 
123     vector<double> numParticles;
124     int accepted = 0;
125     int accepted_1k = 0;
126     int rejected_1k = 0;
127     for (int i=0; i<=(L/width); i++) {
128         numParticles.push_back(0);
129     }
130 
131     for (int i = 0; i<num_iterations; i++) {
132 
133         //Self correcting acceptance rate to keep within 30-70%. Adjusts by factor of
                dr/10, checking each 100000 iterations
134         if (i % 10 == 0 && i!= 0) {
135             double acceptance_rate = (double(accepted_1k)/(accepted_1k + rejected_1k))
                    *100;
136 
137             //If too low e.g. for 20%, adjusts by - (2*dr)/5
138             if (acceptance_rate < 30) {
139                 dr = dr - ((dr/10)*(((30-acceptance_rate)/10)+1));
140             }
141 
142             //Reset dr if we get into a local minima stuck position
143             if (acceptance_rate < 2) {
144                 // dr = 0.1;
145             }
146             else if (acceptance_rate > 70) {
147                 dr = dr + ((dr/10)*(((acceptance_rate-70)/10)+1));
148             }
149             //cout << endl << "Acceptance rate: " << acceptance_rate;
150             //cout << endl << "dr: " << dr;
151             accepted_1k = 0;
152             rejected_1k = 0;
153 
154         }
155 
156         //Calculate probability of current system
157         complex<double> psi1 = createWaveFunction(R1);
158         double p = norm(psi1);
159 
160         //Create empty test system
161         vector<Point> R3;
162 
163         //Populate new system R3 with original particles from R1, plus a random
```

```
            displacement of length dr
        for (Point p1 : R1) {
            double phi = (2*M_PI) * dis(gen);
            double x = p1.x() + (dr * cos(phi));
            double y = p1.y() + (dr * sin(phi));

            //If a particle moves outside the box, instead make it reappear out the
                opposite side
            if (x > (L/2)) { x = x - L;}
            if (x < (-L/2)) {x = x + L;}
            if (y > (L/2)) { y = y - L;}
            if (y < (-L/2)) { y = y + L;}

            R3.push_back(Point(x,y));
        }

        //Calculate probability of new system
        complex<double> psi3 = createWaveFunction(R3);
        double p_new = norm(psi3);

        //Accept in accordance to Hastings-Metropolis method
        double lambda = min(p_new/p,1.0);
        double alpha = dis(gen);

        //If accept the new configuration
        if (alpha < lambda) {

            accepted += 1;
            accepted_1k += 1;

            R1 = R3;
            psi1 = psi3;

            for (Point p1 : R1) {
                int elementID = floor((p1.length()/width));
                numParticles[elementID] += 1;
            }

        }
        else {
            rejected_1k +=1;
        }

    }

    vector<Point> normNumParticles;
    for (double i=0; i<numParticles.size(); i++) {
        //double area = M_PI * (pow(width*(i+1),2) - pow(width*i,2));
        double radius = width;
        double norm_factor = accepted*R1.size()*radius;
        double normNum = numParticles[i]/(norm_factor);
        Point slice = Point(normNum,width*i);
        normNumParticles.push_back(slice);
    }
    cout << accepted << endl;
```

```
    return normNumParticles;
}

//Method using a Metropolis algorithm to iterate the two systems over random moves
vector<double> runMetropolis (vector<Point> rPoints1, vector<Point> rPoints2, double
    dr, int num_iterations) {

    //Define some counters
    int accepted = 0;
    int accepted_1k = 0;
    int rejected = 0;
    int rejected_1k = 0;
    int swapped = 0;

    //Set up array to keep track of integral summations for a set number of Z values
    complex<double> sum_gx = complex<double>(0,0);
    complex<double> sum_g2x = complex<double>(0,0);

    //Define current system
    vector<Point> R1(rPoints1);
    vector<Point> R2(rPoints2);

    for (int i = 0; i<num_iterations; i++) {

        //Self correcting acceptance rate to keep within 30-70%. Adjusts by factor of
            dr/10, checking each 1000 iterations
        if (i % 1000 == 0 && i!= 0) {
            double acceptance_rate = (double(accepted_1k)/(accepted_1k + rejected_1k))
                *100;

            //If too low e.g. for 20%, adjusts by - (2*dr)/5
            if (acceptance_rate < 30) {
                dr = dr - ((dr/10)*(((30-acceptance_rate)/10)+1));
            }

            else if (acceptance_rate > 70) {
                dr = dr + ((dr/10)*(((acceptance_rate-70)/10)+1));
            }
            //cout << endl << "Acceptance rate: " << acceptance_rate;
            //cout << endl << "dr: " << dr;
            accepted_1k = 0;
            rejected_1k = 0;

        }
        //Output progress in increments of 10%
        if (i % (num_iterations/10) == 0 && i!= 0) {
            cout << double((i*100.0/num_iterations)) << "%␣" << flush;
        }

        //Calculate probability of current system
        complex<double> psi1 = createWaveFunction(R1);
        complex<double> psi2 = createWaveFunction(R2);
        double p = calcJointProb(psi1, psi2);

        //Create empty test system
```

```
269         vector<Point> R3;
270         vector<Point> R4;
271 
272         //Populate new system R3 with original particles from R1, plus a random
                displacement of length dr
273         for (Point p1 : R1) {
274             double phi = (2*M_PI) * dis(gen);
275             double x = p1.x() + (dr * cos(phi));
276             double y = p1.y() + (dr * sin(phi));
277 
278             //If a particle moves outside the box, instead make it reappear out the
                    opposite side
279             if (x > (L/2)) { x = x - L;}
280             if (x < (-L/2)) {x = x + L;}
281             if (y > (L/2)) { y = y - L;}
282             if (y < (-L/2)) { y = y + L;}
283 
284             R3.push_back(Point(x,y));
285         }
286 
287         //Repeat for R4/R2
288         for (Point p2 : R2) {
289             double phi = (2*M_PI) * dis(gen);
290             double x = p2.x() + (dr * cos(phi));
291             double y = p2.y() + (dr * sin(phi));
292 
293             if (x > (L/2)) { x = x - L;}
294             if (x < (-L/2)) {x = x + L;}
295             if (y > (L/2)) { y = y - L;}
296             if (y < (-L/2)) { y = y + L;}
297 
298             R4.push_back(Point(x,y));
299 
300         }
301 
302         //Calculate probability of new system
303         complex<double> psi3 = createWaveFunction(R3);
304         complex<double> psi4 = createWaveFunction(R4);
305         double p_new = calcJointProb(psi3,psi4);
306 
307         //Accept in accordance to Hastings-Metropolis method
308         double lambda = min(p_new/p,1.0);
309         double alpha = dis(gen);
310 
311         //If accept the new configuration
312         if (alpha < lambda) {
313 
314             accepted += 1;
315             accepted_1k += 1;
316             complex<double> g;
317             vector<int> iListR3;
318             vector<int> iListR4;
319 
320             //The new moved system becomes our current system
321             R1 = R3;
```

```
322             psi1 = createWaveFunctionTerm(R1, 0);
323             R2 = R4;
324             psi2 = createWaveFunctionTerm(R2, 0);
325 
326             //We will keep using R3 and R4 as the system for our swap calculation
327 
328             //Find the ID of particles within subsystem A (ie. particles left of the y
                    axis)
329             for (int j=0; j<R3.size(); j++) {
330                 if (R3[j].x() < 0) {
331                     iListR3.push_back(j);
332                 }
333                 if (R4[j].x() < 0) {
334                     iListR4.push_back(j);
335 
336                 }
337             }
338             //If the number of particles in R3_A = R4_A we can perform the swap
339             if (iListR3.size() == iListR4.size()) {
340                 swapped += 1;
341                 for (int d=0; d<iListR3.size(); d++) {
342 
343                     //Find the index integer of the particles we are swapping
344                     int id_1 = iListR3[d];
345                     int id_2 = iListR4[d];
346 
347                     //Swap the particles by copying each others coordinates
348                     //At this point R2 is still an exact copy of R4. So in fact we copy
                            R2_A into R3_A to avoid creating temporary copy variables
349                     //And we do the same copying R1_A into R4_A
350                     R3[id_1].copy(R2[id_2]);
351                     R4[id_2].copy(R1[id_1]);
352                 }
353                 //Recalculate the partial wavefunctions of R1,R2,R3,R4 (after the swap)
354 
355                 psi3 = createWaveFunctionTerm(R3,0); //We only need the partial WF as
                        the exponential terms cancel
356                 psi4 = createWaveFunctionTerm(R4,0);
357                 g = (psi3 * psi4)/(psi1 * psi2);
358 
359             }
360             //If we didnt perform the swap, dont add to the integral (dirac delta
                    factor)
361             else {
362                 g = 0;
363             }
364             sum_gx += g;
365             sum_g2x += pow(g,2);
366 
367             iListR3.clear();
368             iListR4.clear();
369 
370         }
371         //Else reject the new configuration
372         else {
```

```
            rejected +=1;
            rejected_1k += 1;
        }

    }
    //Calculate the error of each S2 point
    complex<double> normComplexIntegral = sum_gx / double(accepted);
    double normIntegral = normComplexIntegral.real();

    double normSquaredIntegral = sum_g2x.real() / accepted;

    double stdevS2 =(pow(normSquaredIntegral - pow(normIntegral,2),0.5))/(pow(accepted
        ,0.5)*normIntegral);

    double s2 = -log(normIntegral);

    //Print properties of run
    cout << endl << "Number␣of␣accepted␣moves:␣" << accepted;
    cout << endl << "Number␣of␣rejected␣moves:␣" << rejected;
    cout << endl << "Acceptance␣rate:␣" << (double(accepted)/(accepted + rejected))
        *100 << "%";
    cout << endl << "Final␣dr:" << dr;
    cout << endl << "Number␣of␣swaps:␣" << swapped;

    //Print normalised integral and calculate S2 for each one
    cout << endl << "Normalised␣complex␣integral:␣";
    cout << normComplexIntegral;


    //Print S2 with the standard deviation
    cout << endl << "S2␣value:␣";
    cout << s2 << "+/-" << stdevS2 << endl;

    //Write results to file for plotting

    vector<double> s2Point;
    s2Point = {s2, stdevS2, double(R1.size())};

    return s2Point;
}

//Method to iterate runMetropolis over multiple N values and write it to file
void iterateOverN (int min_n, int max_n, double dr, int num_iterations) {
    vector<vector<double>> s2Points;
    for (int n=min_n; n<max_n+1; n++) {
        vector<Point> R1 = initialiseSystem(n);
        vector<Point> R2 = initialiseSystem(n);
        runBurnIn(R1, R2, 0.1, 1000000);
        cout << endl << n << endl;
        vector<double> s2Point = runMetropolis(R1, R2, dr, num_iterations);
        s2Points.push_back(s2Point);
    }
   string file_name = "MC_n" + to_string(max_n) + "_" + to_string(num_iterations
       /1000000) + "m_L" + to_string(int(L)) + "_m3.txt";
   writeMCToFile(s2Points, file_name);
```

```
}

vector<vector<double>> iterateDensityProfile(int n, int n_max) {

    vector<vector<double>> r0List;

    for (int j=n; j<n_max+1; j++) {
        cout << j << endl;
        double num_iterations = 10000000;
        vector<Point> R1 = initialiseSystem(j);
        vector<Point> R2 = initialiseSystem(j);
        runBurnIn(R1, R2, 0.1, 1000000);
        vector<Point> density = densityProfile(R1, 0.1, num_iterations, 0.1);

        //If plotting total density profile
        string file_name = "density_n" + to_string(j) + "_width" + to_string(density
            [1].y()) + "_L" + to_string(L) + "_" + to_string(num_iterations/1000000) +
            "m.txt";
        writeToFile(density, file_name);

        //If plotting rho_0 or r_0
        /*Point r0 = Point(0,0); //initialise
        Point rho_max = Point(0,0);
        double rho_target = density[0].x();

        //Find maximum point
        for (Point p : density) {
            if (p.x() > rho_max.x()) {
                rho_max = p;
            }
        }

        for (Point p : density) {
            if (abs(p.x()-rho_target) < abs(r0.x()-rho_target) && p.y() > rho_max.y())
                {
                r0 = p;
            }
        }
        vector<double> r0Point = {double(j),r0.x(),r0.y()};
        r0List.push_back(r0Point);*/
    }

    return r0List;

}

int main(int argc, const char * argv[]) {

    iterateOverN(2, 5, 0.1, 10000000);

    //iterateDensityProfile(2, 20);

    /*vector<Point> r0List;
    vector<Point> rho0List;
    vector<vector<double>> r0PointList = iterateDensityProfile(16,20);
```

```
477     for (vector<double> d : r0PointList) {
478         r0List.push_back(Point(d[0],d[2]));
479         rho0List.push_back(Point(d[0],d[1]));
480     }
481     writeToFile(r0List, "r06_vs_n_005.txt");
482     writeToFile(rho0List, "rho06_vs_n_005.txt");*/
483
484
485
486 }
```