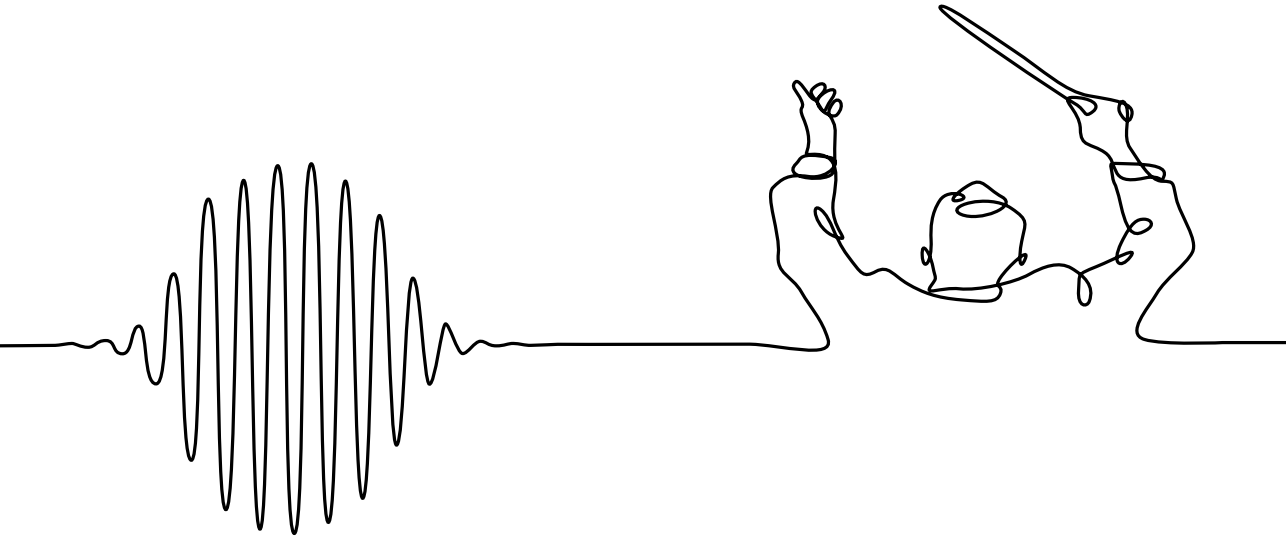
Manual

Free-Fermionic Hilbert Space Search

Daniel M. Chernowitz



2021

Chapter 1

Physical Setup: The Aharonov-Bohm Quench

This manual is a modified excerpt from the PhD thesis of the Daniel M. Chernowitz. Part II of the thesis studies the τ-function. More background and details can be found in the full text at `https://pure.uva.nl/ws/files/68319027/Thesis.pdf` or in the precursor publication [1].

This manual illustrates an application of a form-factor expansion of an operator in a free-fermionic Hilbert space. In our case, the operator is the τ-function, which is highly non-local in momentum-space. Thus it is a good testing ground to attempt to obtain thermal expectation values, where the non-locality conspires with the entropic occupation of quantum numbers. Then exponentially large numbers of states contribute significantly to the operator expectation. If we can obtain the thermal τ-function, we can obtain any operator.

We illustrate a simple physical setup where the τ-function appears naturally: a magnetic quench in the system that exhibits the *Aharonov-Bohm* effect [2]. Namely, consider a continuous one-dimensional circular loop of length L in the horizontal plane, with noninteracting fermions living on it. These could be, for instance, electrons in a wire, all in a spin-up state. There is no spin-flip mechanism in this model, so they are effectively spinless, but they must be able to respond to a magnetic field. Through the loop, this homogeneous vertical magnetic field of strength $\mathcal{B}$ flows, perhaps produced by a solenoid coil. The field only extends to radius $r_c < \frac{L}{2\pi}$, as in figure 1.1.

The resulting electromagnetic vector potential, in cylindrical coordinates (r, z, φ)

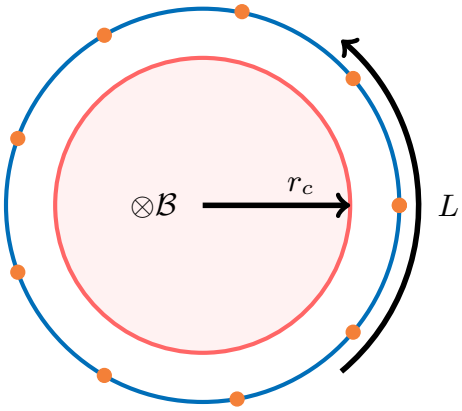


Figure 1.1: Setup for the Aharonov-Bohm effect. A cylindrical region in the center has a constant magnetic field $\mathcal{B}$ going into the page. Its circular intersection is shaded red. Strictly outside it, our loop of circumference L lies in the horizontal plane of the page. On the loop, a number of fermions are schematically indicated in orange.

is as follows:

$$\mathcal{A}_r = \mathcal{A}_z = 0, \quad \mathcal{A}_\varphi = \begin{cases} \frac{\mathcal{B}r}{2}, & r \leq r_c \\ \frac{\mathcal{B}r_c^2}{2r}, & r > r_c. \end{cases} \tag{1.1}$$

The fermions couple to the field with coupling strength e_c. We change coordinates to the position on the loop $x = \frac{L}{2\pi}\varphi$. Due to the magnetic flux $\Phi = \pi r_c^2 \mathcal{B}$, the fermions experience the following Hamiltonian:

$$H = \frac{1}{2m_e}\left(-i\hbar\frac{\partial}{\partial x} - \frac{e_c\Phi}{L}\right)^2. \tag{1.2}$$

The single particle eigenstates of this system are plane waves, with quantized momentum (wavenumber) k. The quantization follows from the periodic boundary condition

$$e^{ikL} = e^{-i2\pi\phi}, \tag{1.3}$$

the RHS of which is the phase picked up by a particle traveling once around the loop. We have defined a parameter

$$\phi := \frac{2\pi\Phi e_c}{\hbar}, \tag{1.4}$$

and tune it to lie on the interval[1] $\phi \in [0, \frac{1}{2})$. This constant, non-integer momentum shift is important, as it interpolates between a trivial free-fermionic problem and a globally coupled one.

We are witness to the conventional Aharonov-Bohm effect: the fermions are not in contact with a magnetic field, but nonetheless by orbiting an area with a vector

[1] Any other $\phi \in \mathbb{R}$ can be found by translation and reflection symmetry.

potential, their path has a nonzero flux, and they pick up a phase. This phase can be observed by means of interference effects, which is a milestone result in topological physics. Moreover, it proves the vector potential to be the fundamental physical object mediating electromagnetic effects.

In the position representation, the wavefunctions of the fermions are

$$\psi(k;x) = \frac{1}{\sqrt{L}} e^{ikx}, \quad k = \frac{2\pi}{L}\left(n - \phi\right), \quad n \in \mathbb{Z}. \tag{1.5}$$

The energy of a single fermion is $E(k) = \frac{\hbar^2}{2m_e}k^2$, and in the following we work in units in which $\hbar^2 = m_e$, so $E = k^2/2$.

Usually, in such setups, the field strength $\mathcal{B}$ is fixed and we are not interested in many body effects. We change this perspective. We consider N of these fermions together without interactions. Due to the Pauli exclusion principle however, they have distinct momenta, collected in a vector $\mathbf{k} := \{k_1 < \ldots < k_N\}$, and their many body wavefunction is a Slater determinant

$$|\mathbf{k}\rangle = \frac{1}{\sqrt{N!}} \det_{1\leq a,b\leq N} \psi(k_a; x_b). \tag{1.6}$$

The multi-particle energy and momentum are the sums of their single particle values.

Furthermore, the system is prepared with zero magnetic field, such that also $\phi = 0$, and at the start of our virtual experiment, $\mathcal{B} > 0$ is switched on, quenching the system to a new eigenbasis with $\phi > 0$. In other words, dynamics suddenly begin according to the new Hamiltonian, while the system finds itself in a thermal ensemble[2] of eigenstates of the original Hamiltonian[3]. When possible, symbols j, g and derivates such as $\mathbf{g}, g_a$ refer to wavenumbers with $\phi = 0$ and k, h to those with $\phi > 0$.

The τ-function may be viewed as a generalized *Loschmidt echo* [3], a kind of correlation function: an expectation value of a translation operator. Such an expectation value is signified by brackets $\langle \ldots \rangle$, and refers to the ensemble, given by context. The τ-function depends explicitly on space and time (x, t), and implicitly on system length L, shift ϕ, temperature T, and on particle number N directly or through the chemical potential, μ.

For an appropriate Hamiltonian $H_\bullet$ and momentum operator $P_\bullet$, $e^{-iH_\bullet t}$ translates the state in time by t and $e^{iP_\bullet x}$ in space by x. In terms of these translation operators, the τ-function is the following

$$\tau(x,t) := \left\langle e^{-iH_\phi t + iP_\phi x} e^{iH_0 t - iP_0 x} \right\rangle, \tag{1.7}$$

where H_ϕ and P_ϕ are the quenched Hamiltonian and momentum operator, which commute. The subscript indicates that here the field is engaged. The pair H_0, P_0 are the same for zero field, and are generally diagonalized by the state in which the system begins. Their inclusion is a gauge choice, but one which will prove to be natural later

[2] See the chapter on the Orthogonality Catastrophe of [1]

[3] Note that this ensemble, for $T = 0$, has all its weight in the unique ground state.

on. The modulus of $\tau(0,t)$ in some generic state $|\mathbf{g}\rangle$ is equal to a Loschmidt echo, defined as the RHS of

$$|\tau_{\mathbf{g}}(0,t)|^2 = \left|\langle \mathbf{g}|\, e^{-itH_\phi} e^{itH_0}\, |\mathbf{g}\rangle\right|^2 . \tag{1.8}$$

Thus the τ-function may be thought to measure recurrence of fidelity to the initial state after a time t, while also rotating the loop onto itself over a distance x. Ensemble averages are obtained by convex addition of the constituent states due to linearity of expectations.

1.1 Definition

To be explicit, the τ-function here is formally defined as an infinite series. Let the state of the system $|\mathbf{g}\rangle$ be specified by N distinct momenta $g_a = \frac{2\pi}{L} n_a$, for integers n_a. Then

$$\tau_{\mathbf{g}}(x,t) := \sum_{\mathbf{k}} |\langle \mathbf{g}|\mathbf{k}\rangle|^2 \, e^{\sum_a \left(ix(g_a - k_a) - it(g_a^2 - k_a^2)/2\right)}. \tag{1.9}$$

This can be obtained from definition (1.7) by inserting a resolution of unity in the eigenbasis $|\mathbf{k}\rangle\langle\mathbf{k}|$ of H_ϕ. The set $\mathbf{k}$ is specified by the N shifted momenta $k_a = \frac{2\pi}{L}(m_a - \phi)$. This way, the infinite summation over available $\mathbf{k}$ is carried out over all ordered sets $\{m_a\}$ of N distinct integers. The final ingredient is the form-factor: in our case the overlap between a shifted and unshifted state. From (1.6),

$$\begin{aligned} \langle \mathbf{g}|\mathbf{k}\rangle :=& \frac{1}{N!}\epsilon_{a_1,\ldots,a_N}\epsilon_{b_1,\ldots,b_N} \int_0^L \prod_{m=1}^N dx_m\, \psi^*(g_{a_m}; x_m)\psi(k_{b_m}; x_m) \\ =& \det_{1\le a,b\le N} \left(\int_0^L dx\, \psi^*(g_a; x)\psi(k_b; x) \right). \end{aligned} \tag{1.10}$$

The second equality uses the simultaneous expansion of the determinant over columns and rows, which allows the integral to factorize over m. The elements of the final determinant are single particle overlaps. From (1.5),

$$\int_0^L dx\, \psi^*(g;x)\psi(k;x) = \frac{1}{L}\int_0^L dx\, e^{i(k-g)x} = \frac{1}{iL(k-g)}\left(e^{-i2\pi\phi} - 1\right) \tag{1.11}$$

using quantization conditions $e^{igL} = 1$ and $e^{ikL} = e^{-i2\pi\phi}$. Then finally, the modulus of the overlap is given by

$$|\langle \mathbf{g}|\mathbf{k}\rangle| = \left(\frac{2}{L}\sin(\pi\phi)\right)^N \left| \det_{1\le a,b\le N} \frac{1}{k_a - g_b} \right|. \tag{1.12}$$

The expression above illustrates the appeal of this model: in other one-dimensional (integrable) many body theories, many observables of interest have been formulated

in terms of a similar determinant. And many more operators can be formally posed in the shape of such a form-factor summation over overlaps times operator values. The specific identities for the overlaps or prefactors may change, but the structure of the sum is quite general.

We believe this toy model and operator have enough structure to be interesting as a stepping stone towards, for instance, the Lieb-Liniger Bose gas, while still being exactly solvable. The choice of form-factor was inspired by and in fact exhibits Anderson's *Orthogonality Catastrophe* (OC).

When approximating the τ-function by computer, we can never perform a full sum over the infinite set of all $\mathbf{k}$. In that case, our approach involves judiciously choosing the terms that collectively hold as much of the weight, or overlap, as possible. Specifically,

$$\tau_{\mathbf{g}}(0,0) = \sum_{\mathbf{k}} |\langle \mathbf{g}|\mathbf{k}\rangle|^2 = 1 \tag{1.13}$$

and truncating the sum results in some measure in $s \in [0,1)$ indicating the quality of our approximation. We refer to this quantity as the *sum rule*. The sum rule will be a guiding statistic in the numerical endeavors of chapter 2.

1.2 The Bose Gas

As an aside, we mention shortly the connected model of Lieb-Liniger, and the Tonks-Girardeau gas. For an elaboration, see [4]. The Tonks-Girardeau gas is the hard core limit of the Lieb-Liniger model of repulsive delta-interacting bosons. A short summary of the Lieb-Liniger model is in order [5].

The gas consists of bosons living on the continuum. Each boson acquires a label a and has coordinate x_a. The Hamiltonian is printed in first quantized form,

$$H = -\sum_{a=1}^{N} \frac{\partial^2}{\partial x_a^2} + 2c \sum_{a<b} \delta(x_a - x_b), \tag{1.14}$$

writing $c \geq 0$ for the interaction strength. In second quantized form,

$$H = \int_0^L dx \partial_x \Psi^\dagger \partial_x \Psi + c\Psi^\dagger \Psi^\dagger \Psi \Psi, \tag{1.15}$$

where Ψ is the bosonic field operator, obeying canonical bosonic commutation relations.

The presence of interaction energy means the particles are in general not free, however, eigenstates are still in bijection to the sets of disjunct integers $\{n_a\}$ for N odd, and half-integers for N even. In lieu of pure single particle momenta, the particles now have sets of *rapidities* k_a that satisfy coupled *Bethe Equations*,

$$Lk_a = 2\pi n_a - 2\sum_{b=1}^{N} \arctan\left(\frac{k_a - k_b}{c}\right). \tag{1.16}$$

We characterize the solutions as plane waves inside any one simplex (an ordered set of positions $\mathbf{x} := \{x_1 < x_2 < \ldots < x_N\}$). Across the boundaries to other simplices, i.e. $x_a = x_b$, $a \neq b$, the waves have cusps due to particle exchange interaction. The position representation is

$$\psi(\mathbf{x}) = \sum_{\sigma} \prod_{1 \leq a < b \leq N} \left(1 + \frac{ic}{k_{\sigma_a} - k_{\sigma_b}}\right) \exp\left\{i \sum_{a=1}^{N} k_{\sigma_a} x_a\right\}, \tag{1.17}$$

which diagonalizes H as in (1.14). The first summation is over the symmetric group of permutations σ. The rest of the spatial domain, non-ordered $\mathbf{x}$, is found by the demand that bosonic wavefunctions are symmetric under any exchange $x_a \leftrightarrow x_b$. This is the *coordinate Bethe Ansatz*.

Total momentum remains unchanged as $\sum_a k_a$ and energy is doubled by convention, $\sum_a k_a^2$.

In the Lieb-Liniger model, there are determinant style operators of central interest. In second quantized notation, in terms of Ψ, typical objects of study are the field-field correlation function,

$$O_1(x,t) := \left\langle \Psi^\dagger(0,0)\Psi(x,t) \right\rangle, \tag{1.18}$$

and the density-density correlation function

$$O_2(x,t) := \left\langle \Psi^\dagger(0,0)\Psi(0,0)\Psi^\dagger(x,t)\Psi(x,t) \right\rangle. \tag{1.19}$$

The form-factors used are single entries of matrices, i.e. for the row corresponding to $\langle \mathbf{g}|$ and column to $|\mathbf{k}\rangle$, we expand in factors $\langle \mathbf{g}|\Psi|\mathbf{k}\rangle$ and $\langle \mathbf{g}|\Psi^\dagger\Psi|\mathbf{k}\rangle$. They, in turn, can be expressed as dressed determinants of the rapidities of the energy eigenstates $\langle \mathbf{g}|$ and $|\mathbf{k}\rangle$. For the precise formulas, which are quite cumbersome and require a number of auxiliary definitions, see expressions (C.1) and (D.1) in the appendix of [5]. Note that for the Bose gas, we must normalize all states explicitly using the Gaudin norm: equation (32) of [5].

With this knowledge, on N-particle state $\langle \mathbf{g}|$, the two-point functions (1.18) and (1.19) can be expanded in a form-factor series by inserting an internal basis of eigenstates $|\mathbf{k}\rangle\langle\mathbf{k}|$, where $\mathbf{k}$ is size N for O_2 and size[4] $N-1$ for O_1. The spacetime dependence is then simply found by evolving according to the translation operators, i.e. multiplying every term by a $\mathbf{k}$ dependent phase. The single state expectation is

$$\langle \mathbf{g}|O_1|\mathbf{g}\rangle = \sum_{\mathbf{k}} |\langle \mathbf{g}|\Psi|\mathbf{k}\rangle|^2 \, e^{i\sum_a \left(x(g_a - k_a) - t(g_a^2 - k_a^2)\right)}, \tag{1.20}$$

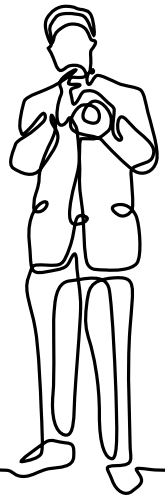
and similar for O_2. These can be used to find ensemble averages by linearity.

As can be seen from the Bethe equations (1.16), for infinite c, the Lieb-Liniger eigenstates have free-fermionic momenta and wave-functions, connecting to the main theme of part **??**. We may adapt the Bethe equations by also shifting $n_a \mapsto n_a - \phi$, termed the *Shifted Bethe equations*, yielding shifted rapidities $\mathbf{k}$. Understanding the

[4] Ψ destroys a particle.

symbols to temporarily represent normalized Lieb-Liniger states and momenta, we can re-interpret the τ-function in (1.9) as an operator on the Lieb-Liniger state[5] with rapidities $\mathbf{g}$.

[5]The only difference is no factor $\frac{1}{2}$ in the energy, and thus this factor missing in front of t in the exponent. We follow the construction of [4], where incidentally τ is called $\mathcal{Q}_N^\kappa(x,t)$ and ϕ is $-i\beta/(2\pi)$.



Chapter 2

Hilbert Space Search Algorithm

In other chapters of part **??**, summations over a multi-fermionic energy eigenbasis were carried out in full, yielding analytic expressions of thermal operators. It would be wise to check these answers.

Even at low system sizes, the fermionic basis is infinite. In practice, by summing a limited subset of terms, an appreciable portion of these operators can already be found. In fact, the error stemming from the curtailed sum can be made arbitrarily small in finite time. That is the perspective of this chapter: we complement the exact expressions with numerical summations that approximate operators such as the τ-function, for finite system sizes, directly from expressions (1.9) and (1.12). This is an entirely independent path and therefore serves as a strong confirmation of those results. Moreover, these numerical techniques can be applied to other systems with fewer mathematical simplifications available. For an example, see section 2.4 where we apply them to the Bose gas.

2.1 Partitioning of Hilbert Space

The first thing that must be understood is the structure of Hilbert space, for we intend to navigate it. In short, the eigenbasis we use is parametrized by finite sets of disjunct integers, as described in section 1. However, 'uniformly' sampling from this set is both unwise and impossible, so we must partition the basis into regimes, which we will later use to carve smart paths that will efficiently produce most of the *weight* of the operators. We will establish nomenclature along the way.

The 'origin' of our new coordinate system of Hilbert space is fixed by choosing a

central state $\langle \mathbf{g}|$, defined by N integers $\{n_b\}$. All other states $|\mathbf{k}\rangle$ will be oriented with respect to $\langle \mathbf{g}|$, based on their similarity to or overlap with it. As a guiding example in this section, we will take

$$\langle \mathbf{g}| \equiv n_b \in \{-7, -5, -4, -1, 0, 1, 2, 5, 9\}. \tag{2.1}$$

The first abstraction is to consider *boxes*, and their *filling*. On the number line, a box is a small integer number B (the box size) of consecutive available single particle quantum numbers, or integers. We give each box an index a. The global state $\langle \mathbf{g}|$ can be classified by the amount of quantum numbers, or fillings $\{\nu_a\}$, occupying each of these boxes. Let $m_{\min}(a)$ be the smallest integer in box a. Then formally,

$$\nu_a := \sum_{b=1}^{N} \sum_{m=m_{\min}(a)}^{m_{\min}(a)+B-1} \delta_{m,n_b}. \tag{2.2}$$

Obviously, $0 \leq \nu_a \leq B$. When we are denoting these box fillings, it is conventional to assume the omitted boxes away from the origin are empty. If we take box size $B = 5$, the box filling of $\langle \mathbf{g}|$ is, from left to right: 1, 3, 3, 2:

$$\langle \mathbf{g}| = \left| \begin{array}{c|c|c|c} -10 & -5 & 0 & 5 \\ \circ\circ\circ\bullet\circ & \bullet\bullet\circ\circ\bullet & \bullet\bullet\bullet\circ\circ & \bullet\circ\circ\circ\bullet \end{array} \right|. \tag{2.3}$$

Here, $\bullet$ indicates an occupied quantum number and $\circ$ and unoccupied one. The numbers above the boxes are $m_{\min}(a)$.

Conversely, given a box filling sequence, we say a *microshuffling* (corresponding to a unique multi-fermion state) consists of a choice of *local shuffling* for each box individually, i.e. placement of the integers in the box. Other microshufflings with the same box filling $\{\nu_a\}$ would be, e.g.

$$\begin{aligned} |\mathbf{k}\rangle &= |\circ\circ\circ\bullet\circ|\bullet\bullet\circ\circ\bullet|\bullet\bullet\circ\circ\bullet|\bullet\circ\circ\circ\bullet| \\ |\mathbf{k}\rangle &= |\bullet\circ\circ\circ\circ|\bullet\circ\bullet\circ\bullet|\circ\circ\bullet\bullet\bullet|\circ\circ\bullet\circ\bullet| \\ |\mathbf{k}\rangle &= |\circ\circ\bullet\circ\circ|\bullet\bullet\bullet\circ\circ|\circ\bullet\bullet\bullet\circ|\circ\circ\bullet\bullet\circ| \\ &\cdots \end{aligned} \tag{2.4}$$

The utility of this language is that it allows us to jointly consider ket states that have a similar distribution of quantum numbers, which in turn strongly predicts the size of an overlap with the reference bra state. Concretely, expressions for $|\langle \mathbf{g}|\mathbf{k}\rangle|$ such as (1.12) and its product identity as a Cauchy determinant show us that overlaps are largest when the distance between the quantum numbers in bra and ket are small, culminating in a maximum overlap, for a given N, which is diagonal.

We immediately introduce a somewhat unintuitive notation, that nonetheless avoids much ambiguity. We index the boxes, starting with index $a = 0$ on the box containing $m = 0$, and then adding boxes alternating from the left and right of those added before[1]. The indices of the boxes in (2.3) are, from left to right, $[3, 1, 0, 2]$. If

[1]This is evocative of a well-known bijection between $\mathbb{N}$ and $\mathbb{Z}$.

more boxes were added to the left, their indices would count up using odd numbers, and to the right using even numbers. If one wishes to know what the index a of a box is, containing quantum number m, the formula is

$$a = \begin{cases} 2|\lfloor m/B \rfloor| & \text{if } m \geq 0 \\ 2|\lfloor m/B \rfloor| - 1 & \text{if } m < 0 \end{cases}. \tag{2.5}$$

Conversely, in order of the box index a,

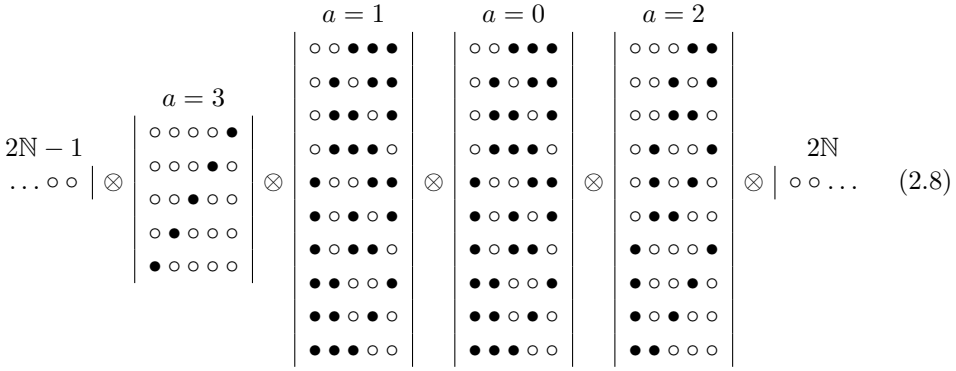
$$m_{\min}(a) \in \{0, -B, B, -2B, 2B, \ldots\} \tag{2.6}$$

When describing boxes, we list them according to their indices, not their numerical ordering. The main advantages are that there is no ambiguity where the counting begins, all boxes will eventually feature at a predictable point, and all trailing omitted numbers in $\{\nu_a\}$ may be assumed zero. This last trait reflects the grouping of quantum numbers near the origin, in turn stemming from the energetic suppression of high momenta. Then the box filling of (2.3) is $\{\nu_0, \nu_1, \nu_2, \nu_3\}_B = \{3, 3, 2, 1\}_5$, subscript holding the box size. We also define the number of *inner boxes*, being the smallest even number M such that

$$\sum_{a=0}^{M-1} \nu_a = N; \quad M \in 2\mathbb{N}, \tag{2.7}$$

so the M inner boxes are symmetrically set around the origin and contain all the quantum numbers of $\langle \mathbf{g}|$.

We can straightforwardly generate all states with a given box filling as the Kronecker product of the spaces of local shufflings over each box. With the origin just to the right of the middle $\otimes$, the filling $\{3, 3, 2, 1\}_5$ represents the set



(2.8)

Although descriptive, the actual box filling of a state $|\mathbf{k}\rangle$ is often less interesting than its *excitation profile* $\{\xi_a\}$ or particle-hole-profile as compared to the reference state. We could say that $\xi_a + \nu_a$ is to $|\mathbf{k}\rangle$ what ν_a is to $\langle \mathbf{g}|$ in (2.2), or $\{\xi_a\}$ is the difference of the box fillings of $|\mathbf{k}\rangle$ with respect to those of $\langle \mathbf{g}|$. It is a sequence of

signed integers, corresponding to the number of *particles* ($\xi_a > 0$) and *holes* ($\xi_a < 0$) created on top of $\{\nu_a\}$:

$$\sum_{a\in\mathbb{N}} \xi_a = 0, \quad 0 \leq \xi_a + \nu_a \leq B, \quad \xi_a \in \mathbb{Z}, \quad \sum_{a\in\mathbb{N}} |\xi_a| = 2\lambda. \tag{2.9}$$

Note that we take the convention $0 \in \mathbb{N}$. This set of demands defines the possible ξ_a, such that $|\mathbf{k}\rangle$ cannot have negative filling or overfull boxes. The last relation introduces a new notion: the *level* of $\{\xi_a\}$, denoted λ. Any excitation profile has a level which is the number of particles ($+1$) as well as the number of holes (-1) created. λ can be thought of as a direction, or axis coordinate in Hilbert space, moving away from the origin $\langle\mathbf{g}|$.

In our example (2.1), any $|\mathbf{k}\rangle$ with box filling $\{4,3,1,1\}_5$ has a level $\lambda = 1$ excitation with excitation profile $\{\xi_a\} = \{+1, 0, -1, 0\}$, while a filling $\{4,3,0,2\}_5$ has a level $\lambda = 2$ excitation, with profile $\{\xi_a\} = \{+1, 0, -2, +1\}$, et cetera. Multiple particles may be in one box, as well as multiple holes, but particles and holes in the same box annihilate.

Moving on, consider the placement of the particles in these excitation profiles. If they are in the M inner boxes, we need not increase the length of the sequence in order to describe the excitation. If, however, they are moved farther out on the number line, we must add a *pad* γ to the excitation, which tracks how many boxes must be added to either side in order to describe the excitation. We always add positive and negative boxes in pairs, and thus always have an even number of elements in the sequences. Let $a_{\max}$ be the largest index such that $\xi_a > 0$, then γ counts such pairs, if any:

$$\gamma := \max\left(0, \left\lceil \frac{a_{\max} - M + 1}{2} \right\rceil\right). \tag{2.10}$$

The pad of composite excitations is the maximum pad among all particles. Comparing to our example (2.3), this next ket state has a $\lambda = 1$, $\gamma = 1$ excitation,

$$\begin{array}{|c|c|c|c|c|c|}
-15 & -10 & -5 & 0 & 5 & 10 \\
\circ\circ\circ\circ\circ & \circ\circ\circ\bullet\circ & \bullet\bullet\circ\circ\bullet & \bullet\circ\circ\circ\bullet & \bullet\circ\circ\circ\bullet & \circ\bullet\circ\circ\circ,
\end{array} \tag{2.11}$$
$$\mapsto \{\xi_a\} = \{-1, 0, 0, 0, +1, 0\}.$$

Recall that the penultimate and ultimate elements of this sequence correspond to the rightmost and leftmost box of the profile, respectively. It is clear that two excitation profiles with different pads cannot contain the same state, so γ forms a partition of Hilbert space: a second direction, or axis.

As a combinatoric exercise, we will often reverse engineer this classification, choose a λ and γ, and produce all $\{\xi_a\}$ that satisfy (2.9) and (2.10). Each of these represents a number of microshufflings. This number may be one, if all $\nu_a + \xi_a \in \{0, B\}$, or it may be very large. It is also possible that there are no $\{\xi_a\}$ available at all, if $\lambda > N$, or if there are fewer than λ possible holes at this γ, i.e. $\lambda > B(M + 2\gamma) - N$.

A final refinement is the concept of *tiers*, using symbols ω and ι. Empirically, varying the exact placement of quantum numbers inside a single box in the ket state

can change the overlap with a reference bra state by an order of magnitude. As more boxes are shuffled locally, this effect compounds, such that even two states with the same box filling can have a virtually vanishing overlap with each other, if their local shufflings are mismatched. Conversely, two states with matching local shuffling on most boxes but a single particle-hole excitation with a high pad might have a larger overlap. In order to account for this hierarchy, we classify local box shufflings, per box a and filling $z = \nu_a + \xi_a$, into tiers $\omega_a(z)$. This object is a finite sequence of sets of local shufflings. The first element of the sequence is tier $\iota_a = 0$, the next tier $\iota_a = 1$, etc. Roughly, the local shufflings in a tier all result in an overlap that is at least[2] some factor $\epsilon < 1$ smaller than those in the next tier down, averaged over the possible shufflings of all the other boxes. The lower the tier, the larger the overlap. The box at index $a = 0$, with filling $z = 3$, might have the following tiers, due to similarity with the local shuffling of $\langle \mathbf{g}|$ in box 0:

$$\omega_0(3) = \left\{ \begin{array}{c} \text{tier 0} \\ \left| \begin{array}{c} \bullet\bullet\circ\bullet\circ \\ \bullet\bullet\bullet\circ\circ \end{array} \right| \end{array}, \begin{array}{c} \text{tier 1} \\ \left| \begin{array}{c} \circ\bullet\bullet\bullet\circ \\ \bullet\circ\bullet\bullet\circ \\ \bullet\bullet\circ\circ\bullet \end{array} \right| \end{array}, \begin{array}{c} \text{tier 2} \\ \left| \bullet\circ\bullet\circ\bullet \right| \end{array}, \begin{array}{c} \text{tier 3} \\ \left| \begin{array}{c} \circ\circ\bullet\bullet\bullet \\ \circ\bullet\circ\bullet\bullet \\ \bullet\circ\circ\bullet\bullet \\ \circ\bullet\bullet\circ\bullet \end{array} \right| \end{array} \right\}. \tag{2.12}$$

The interpretation is that, e.g. (in units of $\frac{2\pi}{L}$),

$$\frac{|\langle \mathbf{g}|\ldots,0,1,3,\ldots\rangle|^2}{|\langle \mathbf{g}|\ldots,0,1,2,\ldots\rangle|^2} = \mathcal{O}(1); \quad \frac{|\langle \mathbf{g}|\ldots,0,1,4,\ldots\rangle|^2}{|\langle \mathbf{g}|\ldots,0,1,2,\ldots\rangle|^2} = \mathcal{O}(\epsilon), \tag{2.13}$$

where the implicit quantum numbers ... are the same above and below the division line. In contrast to (2.12), $\omega_1(3)$, for the box to the left, will likely be completely differently divided. By the same token, when box 0 is excited, $\xi_0 \neq 0$, and has $z \leq 2$ or $z \geq 4$ particles, a new set of tiers must be constructed. Tier 0 always exists, and may only contain the trivial shuffling $|\circ\circ\circ\circ\circ|$ if the filling $z = 0$ or $|\bullet\bullet\bullet\bullet\bullet|$ if $z = B$. Higher tiers may or may not exist, but the union of tiers is always all possible local shufflings. In the case $B = 5$, $z = 3$, there are $\binom{B}{z} = 10$.

It is worth stressing that, contrary to level and pad, the population of $\omega_a(z)$ is not a priori obvious and is a computational choice made on-the-go, based on incomplete information. The specific method is explained in the next section, and is guided by the overlap with the reference state of special subsets of ket states.

Moving forward, we assume for each box a and local filling $0 \leq z \leq B$ of that box, $\omega_a(z)$ have been constructed. Now we can augment the definition to *global tiers*

$$\iota = \sum_{a\in\mathbb{N}} \iota_a. \tag{2.14}$$

For a given microshuffling, we can determine its global tier as follows: For each box a, determine in which tier ι_a its current local shuffling is found, and sum these values.

[2]We choose not to have empty tiers, although the factor between tiers might be $< \epsilon^2$.

This is the reason for starting tiers at 0: the highest weight (tier 0) local shufflings do not contribute to the global tier for any box. Notably, in our notation, we suppress the infinite sequence of empty boxes to the left and right of the origin. These all have the trivial $z = 0$ filling, and thus only tier 0, so their inclusion in the sum does not affect the global tier, allowing it to be well-defined. There is a maximum global tier, which is the sum of the maximal tiers of all boxes at a given excitation profile. Let $\#\omega_a(z)$ be the highest tier in the sequence $\omega_a(z)$. In the example of (2.12), $\#\omega_0(3) = 3$. Then the maximum global tier at $\{\xi_a\}$ is

$$\#\iota(\{\xi_a\}) := \sum_{a\in\mathbb{N}} \#\omega_a(\nu_a + \xi_a). \tag{2.15}$$

Conversely, after determining an excitation profile $\{\xi_a\}$ and with access to $\omega_a(z)$, we can construct all microshufflings that have a certain global tier ι. We first allocate local tiers ι_a such that $\iota_a \leq \#\omega_a(\nu_a + \xi_a)$ for each a, and (2.14) holds. Then we only select local shufflings from the corresponding ${\iota_a}^{\text{th}}$ set of $\omega_a(\nu_a + \xi_a)$. The set of microshufflings with $\iota > \#\iota(\{\xi_a\})$ is empty.

We realize local tiers are a kind of particles in their own right. We can place (excite) local tiers combinatorically in any box, as long as the box has enough tiers to offer. If, for the example of some highly excited profile $\{\xi_a\}$,

$$\{\#\omega_a(\nu_a + \xi_a)\} = \{2, 1, 1, 0\}, \tag{2.16}$$

and we desire a global tier of $\iota = 3$, the local tiers can be allocated as follows:

$$\iota = 3 \Leftrightarrow \{\iota_a\} \in \Big\{\{2, 1, 0, 0\}, \{2, 0, 1, 0\}, \{1, 1, 1, 0\}\Big\}. \tag{2.17}$$

Going back to the example of $\omega_0(3)$ in (2.12), if $\xi_0 = 0$ and $\iota_0 = 2$ we would only be allowed to choose $|\bullet \circ \bullet \circ \bullet|$, but if $\iota_0 = 3$, we have 4 local shufflings from which to choose. Unsurprisingly, ι is the third coordinate axis along which to delineate a $|\mathbf{k}\rangle$-state's position.

We say Hilbert space is partitioned into *blocks*: a block is a set of microstates corresponding to a particular level λ, pad γ and global tier ι. We expect that increasing any of (λ, γ, ι) shrinks $|\langle\mathbf{g}|\mathbf{k}\rangle|^2$. The level and pad comprise many excitation profiles $\{\xi_a\}$, and when given a global tier, each $\{\xi_a\}$ represents a number, or *salvo* of states. The union of all these salvos forms the block. The three characteristics or dimensions aid in the visualisation of it being a cube. When exploring Hilbert space, they neighbor each other incrementally, like a three-dimensional Young diagram. This is captured in diagrams such as figure 2.1.

In conclusion, navigation of free-fermionic Hilbert space is done as follows. The preparation:

- Choose a reference state $\langle\mathbf{g}|$ with specific quantum numbers.
- Divide the number line into boxes of size B and determine their reference filling $\{\nu_a\}$.

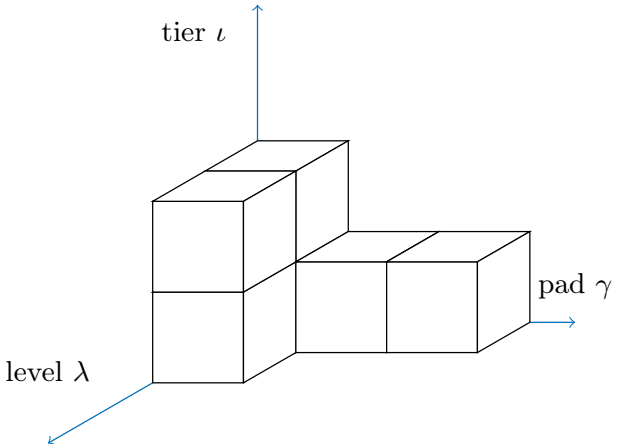


Figure 2.1: The three-dimensional Young diagram partitioning of free-fermionic Hilbert space. We use such diagrams to visualize different secotrs of the space. Each block is defined by a level, pad and tier, and represents a set of states, disjunct from other blocks.

- Construct local tiers $\omega_a(z)$ for each box a and for each filling $z \in \{0, 1, \ldots, B\}$ of that box.
- Then, repeatedly, a set or salvo of states in Hilbert space is found by the steps:
  - Choose a block or triplet $(\lambda, \gamma, \iota) \in \mathbb{N}^3$: an excitation level, a pad for the box of the outer excited particle, and a global tier.
  - Choose an excitation profile $\{\xi_a\}$ corresponding to this pad and level, if available.
  - For every allocation of local tiers $\{\iota_a\}$ that sum to ι, consider all microshufflings at this excitation profile and local tier placement in unison.

The set provided at the end of this protocol is unique and disjunct from the sets found for any other block or excitation profile in this block. We expect all states in a block to have mutually similar overlaps with the reference state. The algorithmic procedure of making all the choices in the above checklist, is described in section 2.2. The goal is an efficient approximation of the τ-function.

On that note: our guiding ansatz is a modification of (1.13). If we only sum a subset $\mathcal{S}$ of the available $|\mathbf{k}\rangle$, as the weight is nonnegative, the result is some number s,

$$s := \sum_{\mathbf{k} \in \mathcal{S}} |\langle \mathbf{g} | \mathbf{k} \rangle|^2 \in [0, 1), \tag{2.18}$$

termed the sum rule. The more weight $|\langle \mathbf{g} | \mathbf{k} \rangle|^2$ can be added, the better. From (1.9), the τ-function is a weighted average of phasors, each with unit modulus. In the

worst case scenario, where they all constructively interfere, the error of the τ-function is exactly the missing weight $1-s$. However, this scenario is only true (modulo periodicity) at $(x,t)=(0,0)$, and at all other points in space-time, the true error is considerably smaller, as the omitted phasors would also destructively interfere. The error satisfies

$$\left|\sum_{\mathbf{k}\notin\mathcal{S}} |\langle\mathbf{g}|\mathbf{k}\rangle|^2 \, e^{\sum_a \left(ix(g_a-k_a)-it(g_a^2-k_a^2)\right)}\right| \leq \sum_{\mathbf{k}\notin\mathcal{S}} |\langle\mathbf{g}|\mathbf{k}\rangle|^2 = 1-s, \tag{2.19}$$

by the triangle inequality. With a judicious protocol to collect $\mathcal{S}$, this error can be expediently minimized.

2.2 Algorithm Search

Now that the structure and nomenclature of the Hilbert space partitioning have been established, we explain at a high level how our algorithm navigates these partitions to efficiently calculate a majority of the weight of the desired operators.

It is evident that inside $\mathcal{S}$, not all $|\mathbf{k}\rangle$ are created equal, but carry different weight. The aim of the algorithm is to incrementally explore more of Hilbert space, expanding the set of blocks that can be chosen, evaluating the contributions of excitation profiles and constructing local tiers progressively. Some preliminaries are needed before we can obtain all states of a block, and we will prepare them in a way that moreover, they yield the 'best states first'. When ready, they can generate states as needed, but will not be immediately depleted. Instead, the algorithm predicts the block with the best weight-to-state ratio and keeps vacillating between blocks to extract states, meanwhile preparing new blocks on the periphery. The scoping out of the tiers is also done during preparation, by generating specific states of the block and assessing their weight. These states need not be revisited later.

When a satisfactory sum rule s is reached, the algorithm terminates. Let us describe the various objects and routines used in the Python code. Functions and important variables are typeset in **bold**. A summary:

Name:	**FFstate**
Purpose:	Class object that approximates the τ-function.
Inputs:	The quantum numbers of the bra state **qn**, the momentum shift **phi**, the box size **B**, and the system length, **L**.
Outputs:	None, however the instance stores many important results as class variables. These are called with the prefix **self.**
Source:	`https://github.com/DMChernowitz/Free-Fermionic-Hilbert-Search`

For those simply wishing to apply or experiment with the routine, it and all dependencies can be found in the 'FFstate.py' file at the URL in the above table. A typical use of the code would be the following:

```
state = FFstate([-7, -4, -3, -2, -1, 0, 1, 2, 3, 4, 6], 0.4, 4, 11)
state.hilbert_search(0.95)
plt.plot(state.tau)
```

This code plots (the real part of) the 11-fermion τ-function, $\phi = 0.4$, with a 0.05 error margin, over $x \in [0, 10]$, for rays $t/x \in \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$.

As an initialization, a reference state object is created, an instance of the class **FFstate**. It has predetermined quantum numbers **qn**, those corresponding to reference state $\langle \mathbf{g}|$. This object is central to this chapter and its subroutines or methods comprise the algorithm. Its box size is the variable **B**. Its filling $\{\nu_a\}$, defined in (2.2), is stored for the M inner boxes. **FFstate** also constructs **self.U**: a nested list, per box, of all possible local shufflings and fillings. At this stage, only the base filling $\{\nu_a\}$ is entered, but later layers of **self.U** will be appended to accommodate excited boxes. For an example of the contents, see expression (A.1). The local fillings are provided by a routine called **ind_draw**, described in appendix A.1, and will be used by other internal subroutines. The τ-function for a discretized mesh of (x, t)-coordinates, is initialized as a complex **Numpy** array with zeros for elements, as no weight has been added yet.

At this point, we are interested in constructing the tiers $\omega_a(\nu_a)$ for the boxes at present filling, without particle-hole excitations. To this end, the first states are produced[3], which comprise all possible microshufflings with the original filling, in the spirit of example (2.8). See figure 2.2 for a diagram of the progress.

This base level shuffling is performed by the **FFstate** method **ground_shuffle** which in turn calls the recursive **in_box_shuffle**, both methods of the **FFstate** class, and the latter described in appendix A.3. In short, it recursively glues local shufflings together in the spirit of equation (2.8). For each state $|\mathbf{k}\rangle$, **ground_shuffle** calls the method **update_tau**, from A.9, where the weight $|\langle \mathbf{g}|\mathbf{k}\rangle|^2$ is calculated, and the contribution to $\tau_{\mathbf{g}}(x, t)$ is stored. **in_box_shuffle** also indexes each local shuffling. This means that each state is returned accompanied by a unique list of indices $\{d_0, d_1, \ldots, d_{M-1}\}$, one index for each of M inner boxes. The utility of this is to allow **ground_shuffle** to fill a high-dimensional tensor termed the **weightrix** with the weights. The dimension of the weightrix is equal to the number of filled boxes, and the size of the a^{th} dimension is the number of local shufflings $\binom{B}{\nu_a}$ in box a. Then each element of the **weightrix** corresponds to exactly one state $|\mathbf{k}\rangle$, and holds the weight $|\langle \mathbf{g}|\mathbf{k}\rangle|^2$. When all weights have been entered, the **weightrix** allows us to construct the first local tiers $\omega_a(\nu_a)$. We find a normalized proxy for the importance of a local shuffling of a box by integrating out the shuffling of all other boxes. If the weightrix is denoted W, then the proxy for box a, shuffling d_a is given by

[3]There is a check, if the *complexity* (calculated in **self.complexity** as the number of unexcited microshufflings) of the bra state is too high, the code will not begin lest it spend too long in preparation. Often, the maximum allowed complexity will be around 400K states.

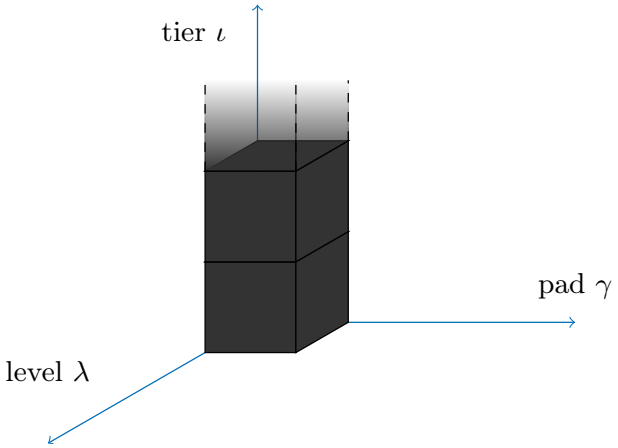


Figure 2.2: Visualization of the progress of the algorithm, when **ground_shuffle** has been executed. The dark column above the origin is level zero, pad zero, and all states, and thus all tiers for zero pad and zero level are treated by this method. None of these need revisiting, so we may think of all these blocks as being depleted and therefore 'inactive'.

$$w_{d_a}(a) = \frac{\sum_{d_0,\ldots,d_{a-1},d_{a+1},\ldots d_{l-1}} W_{d_0,\ldots,d_{l-1}}}{\sum_{d_0,\ldots,d_{l-1}} W_{d_0,\ldots,d_{l-1}}}. \tag{2.20}$$

This is calculated in Python by the routine **dissect**, discussed in appendix A.4. With this proxy, the tiers ι_a for these inner boxes are constructed. The algorithm uses a preset constant $\epsilon = 0.3$ to separate the local shufflings into tiers. This choice of ϵ is found empirically to produce the quickest search in combination with a box size of $B = 4$, and between 10 and 100 particles. The exact division into tiers is performed by subroutine **sortout** from A.5. In short: we attempt to bin their proxy weight $w_{d_a}(a)$ on a log scale into bins of size $\log \epsilon$. Each bin is a local tier.

The $\omega_a(\nu_a)$ are stored in the variable **self.BT** of the **FFstate** class, created by **ground_shuffle**. This is a highly nested list. The first index is the excitation ξ_a (indices may be negative), the second is the box index a, and the third is ι_a. At this depth, one finds a list of local shufflings (themselves lists) of the quantum numbers in that box: these shufflings comprise the tier. **self.BT** will be used continuously in the rest of the search algorithm.

At the same time, we create another variable of the **FFstate** class, **self.p_h_profiles**, with a similar nested structure. The purpose of this variable is to hold any and all admissible excitation profiles. The first index of **self.p_h_profiles** is the level λ, the second is the pad γ, and the third enumerates the possible excitation profiles at this λ, γ. For each such profile, an entry consists of a list of the following three objects: the profile, stored as a list of integers in the style of example (2.11), a

list of the *extremal* indices of this profile, and a *walking*[4] variable for this profile. An extremal index is a box index $\tilde{a}$ such that the box is filled by a number of particles not found at a lower λ or γ for this box. Trivially, an extremal box $\tilde{a}$ must have $\xi_{\tilde{a}} = \pm\lambda$ particles or holes. If $\gamma > 0$, then only the outer two boxes (or last two indices) can be extremal, and only if their box has λ particles, because putting λ particles[5] in lower-indexed boxes occurs at a lower pad already. It should be clear that an excitation profiles has zero, one or at most two extremal indices, one for particles and one for holes. If the particles or holes are divided over more than one box, they don't contribute an extremal index. The utility of the extremal indices is directly tied to the creation of local tiers at nonzero level. They will feature in the following stage of the algorithm: exploration, where in extremal boxes a, all local shufflings are used, in order to construct $\omega_a(\nu_a \pm \lambda)$.

Before terminating, **ground_shuffle** constructs the **self.blocks** and **self.cart** variables for the **FFstate** instance.

self.blocks is another nested list and has first, second and third indices for λ, γ, ι, and holds an instance of **self.deepen_gen** as each element, a generator for salvos (small sets) of states in that block. This generator is explained in A.6, but it is similar to the recursive generation of **ground_shuffle**, except it takes care to allocate $\{\iota_a\}$ as in example (2.17), and avoids the microshufflings already used during exploration. Recall that just because we have prepared the generator, it does not mean we have prompted it to produce all states in the blocks, and most have not yet been evaluated.

The related **self.cart** holds a sorted list of entries, each corresponding to a block, an address in Hilbert space. The intention is that the top entry of **self.cart** is the block where to search for states next, i.e. from which generator in **self.blocks** we expect the most new weight per state to lie. Entries in **self.cart** are each lists, whose first entries are the weight per state, produced over the most recent salvo from an instance of **deepen_gen**, and the rest of the list is the λ, γ, ι of that block, so it can be found in **self.blocks**. An example of a snapshot of **self.cart** is given further down in (2.22).

After **ground_shuffle**, we jump-start exploration with the **build_block** method of the **FFstate** class, which is intended to scout just beyond the boundaries of states already produced. **build_block** is called with the arguments of the current level and pad, at this stage in the algorithm, $\lambda = 0 = \gamma$. **build_block** calls another method named **explore**, which prepares, or explores, a new column of blocks: this means to construct the necessary local tiers and possible excitation profiles in the column of blocks. If the current pad is the maximally explored pad at this level[6], **build_block** calls **explore** at $(\lambda, \gamma + 1)$. If the current level is the maximally explored level, and

[4]This **walking** variable holds the sum of unsquared overlaps $\langle \mathbf{g}|\mathbf{k}\rangle$ of all states $|\mathbf{k}\rangle$ produced from this profile, as the algorithm progresses. This is useful information when trying to understand the field theory limit of these states, where many microshufflings correspond to a single field theory state, labeled by the excitation profile.

[5]We also can't put holes outside the inner boxes, by definition.

[6]It may occur, for instance at level 1, that we call **explore** on a level and pad such that lower levels at this pad have not been explored in this instance of the **FFstate** object, though the resulting tiers from lower levels are needed. Therefore, **explore** first calls itself on one level lower, at this pad.

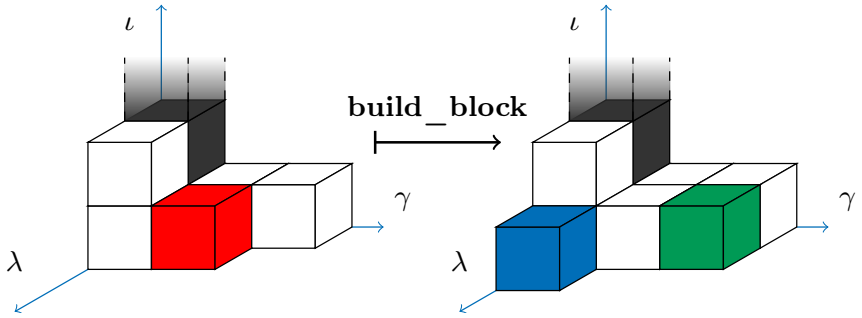


Figure 2.3: Visualization of the three-dimensional Young diagram partitioning of free-fermionic Hilbert space, at some point during the search. The red block is the current block the algorithm uses to produce states. When **build_block** is called, it explores the column of the green block, and if **self.peps** ≤ 1, also the blue, constructing their excitation profiles and the local tiers they use. Then it adds the green (blue) block(s) to **self.cart**.

$\gamma \geq$ **self.peps** $\cdot\, \lambda$, **build_block** calls **explore** at one level higher and zero pad[7]. The variable **self.peps** is usually 1 or 2, and the use of this condition prevents the algorithm exploring high levels before putting low numbers of particles at appreciable distance from the origin, which empirically carry more weight. High level exploration is extremely expensive in computation time, for simple combinatoric reasons. For a schematic of the **build_block** rule, see figure 2.3. Using this procedure, we generally avoid exploring blocks unless we expect them to be used soon after.

The code used in **explore** is printed in A.10. In short, it functions as follows. **explore** produces all states at the new pad and level with a particular characteristic: in extremal boxes, we allow all local shufflings, as the tiers are not yet known. For any excitation profile, most if not all boxes are non-extremal, so their tiers are known, and we only allow tier 0 local shufflings in these. The weights of these states allow us to construct tiers for extremal boxes, again by means of a **weightrix**.

Though carrying the same name as the variable in **ground_shuffle**, the structure of the **weightrix** is quite different at this stage, because we only need to determine tiers for extremal boxes.

In case **explore** is called with a pad of zero, the **weightrix** is a nested list of lists. Let the level be λ. The first index of the **weightrix** takes values 0 or 1 and determines whether we are describing particles or holes, respectively. The second index $a \in \{0, \ldots, M-1\}$ runs over the original inner boxes. The third index d_a enumerates the possible local shufflings in box a when it is extremal: with filling $\nu_a + \lambda$, in the case of particles, or $\nu_a - \lambda$, in the case of holes. There may be no such

[7]It may happen that the level λ is too high to relocate all quantum numbers inside the allowed boxes at a certain pad. There is a fail-safe, until there are at least some states available in the generator of **self.deepen_gen**, **build_block** keeps increasing the pad, up to a maximum pad determined beforehand.

shufflings, if the $\lambda > \nu_a$ for holes or $\lambda > B - \nu_a$ for particles, then that a holds an empty list. The entry of the **weightrix** $W^{(0)}_{d_a}(a)$, is the sum of the weights $|\langle \mathbf{g}|\mathbf{k}\rangle|^2$ of all states $|\mathbf{k}\rangle$ produced by **explore** that have λ particles in box a, while also having the local shuffling corresponding to d_a in that box. The analogous is true for $W^{(1)}_{d_a}(a)$ but with holes. If their excitation profile has an extremal box, states contribute to a term in the **weightrix**, and a few states, for instance with an excitation profile $\{-\lambda, 0, 0, +\lambda\}$, contribute to both $W^{(1)}_{d_0}(0)$ and $W^{(0)}_{d_3}(3)$.

If, on the other hand, **explore** is called with the pad nonzero, then extremal boxes can only be the outer two (those with the highest indices), and only with λ particles in them, not holes. It follows that the **weightrix** is one dimension smaller, we omit the first index, and the box index can only run over the two values $a \in \{M + 2\gamma - 2, M + 2\gamma - 1\}$.

The proxy weight for an extremal box a is simply the entry of W, normalized over the available local shufflings of a, i.e. index d_a. Dropping the superscript, which only applies for zero pad,

$$w_{d_a}(a) = \frac{W_{d_a}(a)}{\sum_{d_a} W_{d_a}(a)}. \tag{2.21}$$

Using these weights, **explore** calls the method **sortout** again, treated in A.5, in order to update the **self.BT** variable with the newly minted tiers for the extremal boxes at this level and pad. It also adds a **deepen_gen** instance at this level and pad, and with input tier zero, corresponding to $\iota = 1$, to the **self.blocks** variable of the **FFstate** instance[8]. With this generator, the rest of the states in the block, not used during exploration, can be yielded.

All the above is executed upon initialization. After the first calls to explore, there are some generators in **self.blocks** and block addresses in **self.cart**. Now the undetermined stage of the algorithm begins.

On our instance of the **FFstate** class, we call the method **hilbert_search**, with two inputs: the desired sum rule, and the maximum run time of the search. If the latter is not given, it defaults to $10N$ seconds. If either are reached, as checked between exploration or salvos of states, the search is terminated. **hilbert_search** has the following structure: While the desired sum rule and time are not met, and the cart is non-empty, repeat the following steps:

- Sort **self.cart** descending by the first entry of each element[9]: the average weight over the last salvo of states from the corresponding block.
- The top block of **self.cart** becomes the current block. With its level, pad and tier:
  - Run **build_block** to explore if necessary.

[8]The $\iota = 0$ states are all amongst those produced by **explore**. The lowest ι that can be prompted from **deepen_gen** is 1.

[9]If the cart is empty, more blocks on the periphery are first explored.

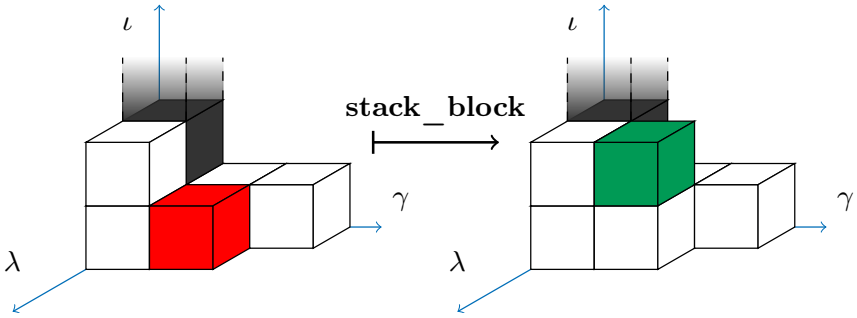


Figure 2.4: Visualization of the three-dimensional Young diagram partitioning of free-fermionic Hilbert space. The first time **stack_block** is called on the current block (shown in red), it adds the green block to **self.cart** and **self.blocks** in the **FFstate** instance, so higher tier states may be produced at this level and pad.

  - Run **stack_block** to add more tiers if necessary.

- If the previous two steps added any blocks tot the **self.cart**, we restart the loop, otherwise, we run the method **harvest**.

During the **hilbert_search** stage, **self.cart** could have the following contents:

$$\textbf{self.cart} = \left[\begin{array}{ccc} \langle w \rangle & & [\lambda, \gamma, \iota] \\ \hline \left[\ 0.013 \ , \ [1,1,0] \ \right], \\ \left[\ 0.009 \ , \ [1,0,2] \ \right], \\ \cdots \quad , \quad \cdots \\ \left[\ 0.006 \ , \ [2,0,1] \ \right] \end{array}\right]. \tag{2.22}$$

If the current block is atop its column, **stack_block** increases its height by adding the next tier **deepen_gen** to **self.blocks**, and its coordinates to **self.cart**. For a schematic representation, see figure 2.4.

In turn, **harvest** extracts salvos of states from the **deepen_gen** generators in the current block in **self.blocks**. When the average weight found dips below the second entry of **self.cart**, we return to the main loop in **hilbert_search**, continuously optimizing our search area. If a block is fully depleted, its address is removed from **self.cart** and the block cannot be current again.

hilbert_search remembers its progress through the use of generators, such that once it terminates, it can be called again on the same **FFstate** instance to add more time and continue the search. The main code of the algorithm is printed below. Some of the **FFstate** class methods, as well as general auxiliary functions, are omitted, those whose code is treated in the appendices A. While running, the algorithm prints pairs of numbers indicating the level, followed by the pad, that are currently being explored.

```
from numpy import zeros, pi, array as arr, sin
from numpy import exp, prod, linspace, newaxis, kron
from numpy.linalg import det
import matplotlib.pyplot as plt
from time import clock

class FFstate:
    def __init__(self,qn,phi,B,L,eps=0.3,ceps=0.2,maxcomplex=200000):
        self.states_done = 0
        self.clo = clock()
        self.N = len(qn)
        self.L = L
        self.B = B
        if B >=self.N:
            print('box too large')
        self.phi = phi
        self.maxord = self.L
        self.qn = qn
        self.abra = arr(self.qn)[:,newaxis]
        shiftqn = arr(self.qn)-phi
        self.pbra,self.ebra = -sum(shiftqn),-sum(shiftqn*shiftqn)
        self.presin = (sin(pi*self.phi)/pi)**self.N
        self.eps = eps
        self.carteps = ceps
        self.peps = 1
        sca = 2*pi/self.L
        xpts = 201
        place = linspace(0,10,xpts)
        rpts = 5
        ray = linspace(0,1,rpts)
        raytime = kron(place,ray).reshape((xpts,rpts))
        self.sum_rule = 0
        self.Place = sca*1j*place[:,newaxis]
        self.Time = -sca*sca*0.5j*raytime #
        self.tau = zeros((xpts,rpts),dtype=complex)
        self.get_box_filling()
        print('box fill:',self.boxfil,'shuffle states:',
              self.complexity,end=', ')
        if self.complexity < maxcomplex and self.complexity>0:
            self.ground_shuffle()
            self.build_block(0,0)
        else:
            print('This would take too long.')

    def qn2box(self,q):
        return 2*abs(q//self.B)-int(q<0)

```

```
48      def box2qn(self,boxj):
49          return (1-2*(boxj%2))*((boxj+1)//2)*self.B
50
51      def local_shuffle(self,fillin,s):
52          locshuf = []
53          for bf in ind_draw(s,s+self.B,fillin):
54              p,e = 0,0
55              for n in bf:
56                  p += n
57                  e += n*n
58              locshuf.append([bf,p,e])
59          return locshuf
60
61      def get_box_filling(self):
62          y = [abs(u//self.B)+int(u>=0) for u in [self.qn[0],self.qn[-1]]]
63          self.innerboxes = 2*max(y+[(self.N+self.B-1)//(2*self.B)+1])
64          self.boxfil = [0 for _ in range(self.innerboxes)]
65          self.maxpad = (self.N//self.B)*5
66          for q in self.qn:
67              self.boxfil[self.qn2box(q)] += 1
68          self.complexity = prod([ncr(self.B,bb) for bb in self.boxfil])
69          self.boxemp = [self.B-bb for bb in self.boxfil]
70          s = 0
71          self.U = [[]]
72          for j in range(self.innerboxes):
73              self.U[0].append(self.local_shuffle(self.boxfil[j],s))
74              if s<0:
75                  s = -s
76              else:
77                  s = -s-self.B
78
79      def ground_shuffle(self):
80          r0 =[ncr(self.B,nnf) for nnf in self.boxfil]
81          weightrix = zeros(shape=tuple(r0))
82          walking = 0
83          for ket,p,e,multind in self.in_box_shuffle(self.boxfil):
84              w,w2 = self.update_tau(ket,p,e)
85              weightrix[tuple(multind)] = w2
86              walking += w
87          odw = dissect(weightrix)
88          self.BT = [[None for boj in range(self.innerboxes)]]
89          self.sortout(odw,0,0)
90          z0 = [0 for _ in range(self.innerboxes)]
91          self.p_h_profiles = [[[[z0,[],walking]]]]
92          self.blocks = [[[]]]
93          self.cart = []
94
```

```
95      def stack_block(self,l,p,t):
96          if len(self.blocks[l][p]) == t+1:
97              self.blocks[l][p].append(self.deepen_gen(l,p,t+1))
98              cm = None
99              for rm,cm in self.blocks[l][p][t+1]:
100                 if cm:
101                     self.cart.append([rm/cm,[l,p,t+1]])
102                     break
103
104     def harvest(self):
105         l,p,t = self.cart[0][1]
106         if len(self.cart)>1:
107             thresh = self.cart[1][0]
108         else:
109             thresh=0
110         for rm,cm in self.blocks[l][p][t]:
111             if rm < thresh*cm:
112                 self.cart[0][0] = rm/cm
113                 return
114         self.cart.pop(0)
115
116     def build_block(self,l,p):
117         if len(self.blocks[l]) == p+1 and p<self.maxpad:
118             self.explore(l,p+1)
119             print(l,p+1,end=' ... ')
120         if len(self.blocks) == l+1 and l<self.maxord and p >= self.peps*l:
121             for p0 in range(self.maxpad+1):
122                 print(l+1,p0,end=' ... ')
123                 self.explore(l+1,p0)
124                 if self.cart and self.cart[-1][1]==[l+1,p0,0]:
125                     return
126             print('not enough tiers')
127
128     def hilbert_search(self,wish,maxtime = 0):
129         if not maxtime:
130             self.clo = self.N*3 + clock()
131         else:
132             self.clo = maxtime + clock()
133         while self.sum_rule < wish and clock()<self.clo:
134             if self.cart:
135                 self.cart.sort(reverse=True)
136                 l,p,t = self.cart[0][1]
137                 lc = len(self.cart)
138                 self.build_block(l,p)
139                 self.stack_block(l,p,t)
140                 if len(self.cart)> lc:
141                     continue
```

```
                self.harvest()
            else:
                l1 = len(self.blocks)
                for l in range(l1):
                    pma = len(self.blocks[l])
                    if pma <= self.maxpad:
                        self.explore(l,pma)
                        l = -1
                        break
                if l1 <= self.maxord:
                    self.explore(l1,0)
                elif l>0:
                    return
        print('!')
```

2.3 Comparison Numerical to Analytic Results

We now show how successful the algorithm is at calculating the τ-function. For thermal expectation values, we must average classically over system states drawn from a thermal ensemble. We use the grand canonical ensemble, choosing single-particle quantum numbers independently to construct multi-particle states. First, we pick a finite system length L. The Fermi-Dirac distribution

$$\rho(q) = \frac{1}{e^{(q^2/2-\mu)/T}+1}, \tag{2.23}$$

is then fitted with chemical potential μ such that the expected filling $\langle N\rangle = L$, however, for finite realizations each may be off by some margin[10]. This means the resulting operator is averaged over states with different particle numbers. In the plots to come, we highlight a different domain of the τ-function than in other chapters: as a function of position x, in the static case ($t = 0$) and at a speed $v = 1$. This features larger amplitudes which are slightly easier to handle numerically. We know by construction $\tau(0,0) = 1$. However, this is also the hardest point to correctly compute by summing basis states: all contributing phasors interfere constructively, and we would need to saturate the sum rule s to recover this result. For this reason, around $x = 0$, the algorithm has the largest error, which is $1 - s$. Away from the origin, the missing phasors are mostly the highly-oscillating ones, and they would cancel each other largely, meaning at most points of space-time, our error is much lower, as evinced by the figure below. We compare the results of the algorithm with the Fredholm determinant from (2.24) below in figure 2.5. The parameters chosen are $T \in \{0.5, 1.5, 3\}$, shift $\phi = 0.4$, and $L \in \{15, 25\}$. The derivation of (2.24) can be

[10]For $L = 25$, the standard deviation is close to 1.6 particles.

found in [1].

$$\begin{aligned}\tau(x,t) =& \det_{\mathbb{R}^2}\left[\mathbb{1}+\hat{K}_\rho\right],\\ K_\rho(p,q) :=& \sqrt{\rho(p)}\frac{e_+(p)e_-(q)-e_-(p)e_+(q)}{p-q}\sqrt{\rho(q)}.\end{aligned} \tag{2.24}$$

For clarity, we repeat the auxiliary dynamic functions

$$\begin{aligned}e_-(q) &= e^{\frac{i}{2}xq-\frac{i}{4}tq^2}\\ e_+(q) &= -\frac{\sin^2(\pi\phi)}{\pi e_-(q)}\left[\cot(\pi\phi)+i\,\mathrm{erf}\left(\frac{(i+1)(x-qt)}{2\sqrt{t}}\right)\right].\end{aligned} \tag{2.25}$$

We see near-perfect agreement outside a neightborhood of $x=0$. It is also clear that, although small, $L=15$ already approaches the TDL in its behavior. Increasing particle number reduces accuracy computationally[11] but does not appear to augment the many-body character of this system. The agreement is especially striking in the static case, likely because the required phasors do not oscillate so rapidly ($\mathcal{O}(k^2t)$), meaning less finely tuned summation is required to achieve high precision. The amplitude for the dynamic case $x=t$ is quite accurate, but mainly the phase suffers. Here 50 samples are used per plot. We expect adding samples would improve the erratic average at larger x.

[11] All plots shown here can be generated on a single CPU of a standard Intel Core i7 3GHz laptop computer in a matter of hours. The typical number of ket states considered varies wildly per bra state, depending on the entropy of the latter. It could be anywhere from several hundred for the bras at $T=\frac{1}{2}, L=15$ to around a million, for the those at $T=3, L=25$.

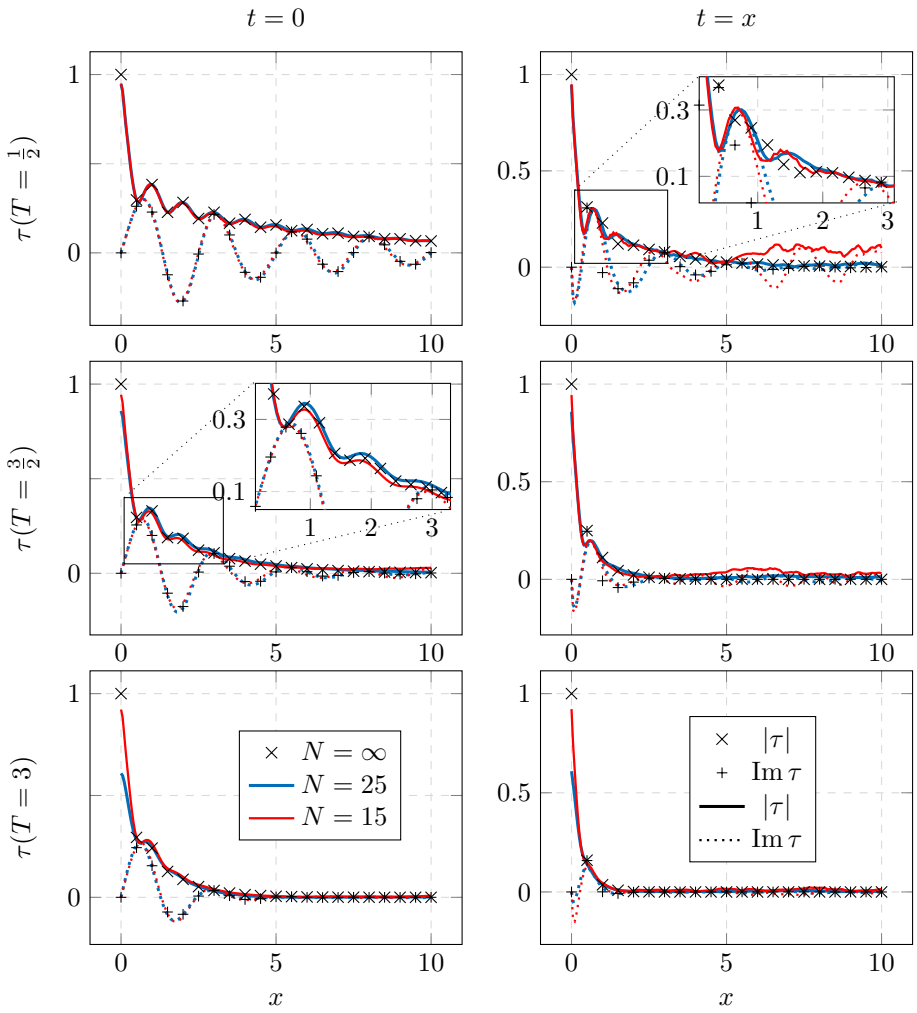


Figure 2.5: Numerical $\tau(x)$ via the algorithm. Blue and red lines are $L = 25, 15$ respectively. μ is such that $\langle N \rangle = L$ in the Gibbs ensemble. Superimposed are the exact Fredholm in black marks: The solid lines (X's) are the envelope $|\tau|$, dotted lines (+'s) are the oscillating imaginary part. Downward, we increase $T \in \{0.5, 1.5, 3\}$. The left column is the static case, right the ray $v = 1$. Box size $B = 4$, shift $\phi = 0.4$. Each plot is a flat average of 50 sampled states.

2.4 Lieb Liniger Model Trial

As a proof of concept, we have modified the code from chapter 2.2 to handle a system with an isomorphic Hilbert space: the repulsive Lieb-Liniger model [6]. This integrable gas of δ-interacting bosons on the continuum was introduced in section 1.2. The main takeaways were the determinant valued operators O_1 and O_2, in (1.18) and (1.19) the field-field and density-density correlators, respectively.

As can be seen from equations (1.20), numerical calculation of these O_1 and O_2 is similar in complexity to that of the τ-function. The largest difference is that the Bethe equations must be solved for every eigenstate, and the Gaudin norm calculated. This is done with the Newton-Raphson method, which incurs the lion's share of the computational cost. A convenient property of the Gaudin matrix (see (33) of [5]) is that it features both as the gradient used to solve (1.16) in Newton's method, as well as the main ingredient of the Gaudin norm. The last iteration of the solver can provide the norm of the Bethe state.

On the other hand, empirically the weight $|\langle \mathbf{g}|\Psi|\mathbf{k}\rangle|^2$ is more concentrated in a smaller number of eigenstates, as compared to the shifted fermion basis (1.12). In other words, the distribution is more skewed over $|\mathbf{k}\rangle$, reducing the number of states that must be visited to shrink the error to a satisfactory value. Hence for similar system sizes, at the analytically most challenging interaction strength of $c \approx 4$ [7], the Lieb-Liniger calculations are comparable in efficiency those of the free-fermionic τ-function (1.9).

One computational speed-up specific to Bethe-solvable models is achieved by computing the Jacobian of the rapidities as a function of the original quantum numbers, and applying a linear correction to the original rapidities in order to approximate the rapidities of states with new integers. Deriving (1.16) to ∂n_b, the Jacobian $\frac{\partial k_a}{\partial n_b}$ is found to satisfy

$$L\frac{\partial k_a}{\partial n_b} + 2\sum_{m=1}^{N}\left(\frac{c}{c^2+(k_a-k_m)^2}\left(\frac{\partial k_a}{\partial n_b}-\frac{\partial k_m}{\partial n_b}\right)\right) - 2\pi\delta_{a,b} = 0, \tag{2.26}$$

which is also solved once, upon initialization, with the Newton-Raphson method. The inputs are the rapidities $\{g_a\}$ of the bra $\langle \mathbf{g}|$ state for the density operator O_2. Because the basis for expanding O_1 has one boson fewer, we excite from a state whose quantum numbers are obtained from those of $\langle \mathbf{g}|$ by removing the integer closest to the origin, and moving all positive integers down by $\frac{1}{2}$, and negative integers up by $\frac{1}{2}$. Accordingly, the solutions $\{k_a\}$ of these inputs to the Bethe equations are used in (2.26) for the field-field correlator O_1.

As each local box shuffling is added recursively in **in_box_shuffle** (see A.3) or **shuf_prof_tier** (see A.8), we can already linearly add the contributions of integers $\tilde{n}_b$, which are displaced from the original n_b, to the predicted rapidities $\tilde{k}_a$, summing over b in bursts in

$$\tilde{k}_a \approx k_a + \sum_b \frac{\partial k_a}{\partial n_b}(\tilde{n}_b - n_b). \tag{2.27}$$

One extra step is necessary because of the choice of box index ordering in (2.5) and the excitations between boxes. In order to match $\tilde{n}_b$ to n_b, at the same b, we must know, for each box, how many occupied quantum numbers are left of that box. This is calculated beforehand from the box filling ν_a and the excitation profile $\{\xi_a\}$, and passed as a list to the modified **in_box_shuffle** and **shuf_prof_tier**.

Thermally distributed states for infinitely large systems are found by sampling single particle quantum numbers according to the solutions $\rho(q)$ of the *Thermodynamic Bethe Equations* [8], exactly as in the free fermion case. For finite systems, the probability of a quantum number being occupied depends on the other occupied numbers, and technically, they cannot be sampled independently. Nonetheless sampling that way is the norm is this field of research. Tabulating a large enough selection of many-body states, computing their energies, and sampling according to the canonical ensemble weight is alas computationally prohibitive. For a selection of the results[12] for $O_1(x,0)$ and $O_1(x,x)$, $N = L = 16$ strictly, see figure 2.6. We have chosen the temperature to be twice that of the comparable plots for the τ-function, to compensate for the energy being doubled as well. We see again that for very entropic bra states, the algorithm struggles to find the full weight $s = 1$, visible around $x = 0$.

As for O_2, we cannot plot in the same way: it is not normal ordered, from which it follows that there is a Dirac delta divergence at $(x,t) = (0,0)$. The sum rule is therefore by definition infinite. Instead, we show what is essentially the Fourier-transform of O_2: the *Dynamical Structure Factor* (DSF) [7]. We bin the weight per contributing state of the operator, according to its momentum and energy, on a heat map, to obtain $O_2(p,E)$. Because the operator is always real, the weight (the form-factor squared) is already positive, and all states accumulate constructively. In order to aid in discerning the weight differences, which span many orders of magnitude, we have in fact plotted the log of the weight density of $O_2(p,E)$. The units of the axis are in the Fermi momentum and energy: $k_\mathrm{F} := h_N$ and $E_\mathrm{F} := h_N^2$ for $|\mathbf{h}\rangle$ the N-particle ground state at $c = 4$. See figure 2.7 for the $N = L = 15, T = 6$ case. These subfigures also offer an insight into which states the operator scopes out: in the white areas, no states have been considered. The algorithm was run until a 'sum rule' of $s = 12$ was attained, any positive number is possible: more and more states at the periphery would be added[13].

[12] The $\langle O_1 \rangle$ values were obtained on a single thread of a laptop computer, with an Intel i7 3Ghz processor, in less than an day.

[13] Multiple $\langle O_2 \rangle$ runs were performed on the same computer in a matter of hours. The average number of ket states $\langle \# \rangle$ visited per $N = 15$ bra state were, for $T \in \{1,3,6\}$ were, respectively, 14K, 128K and 206K.

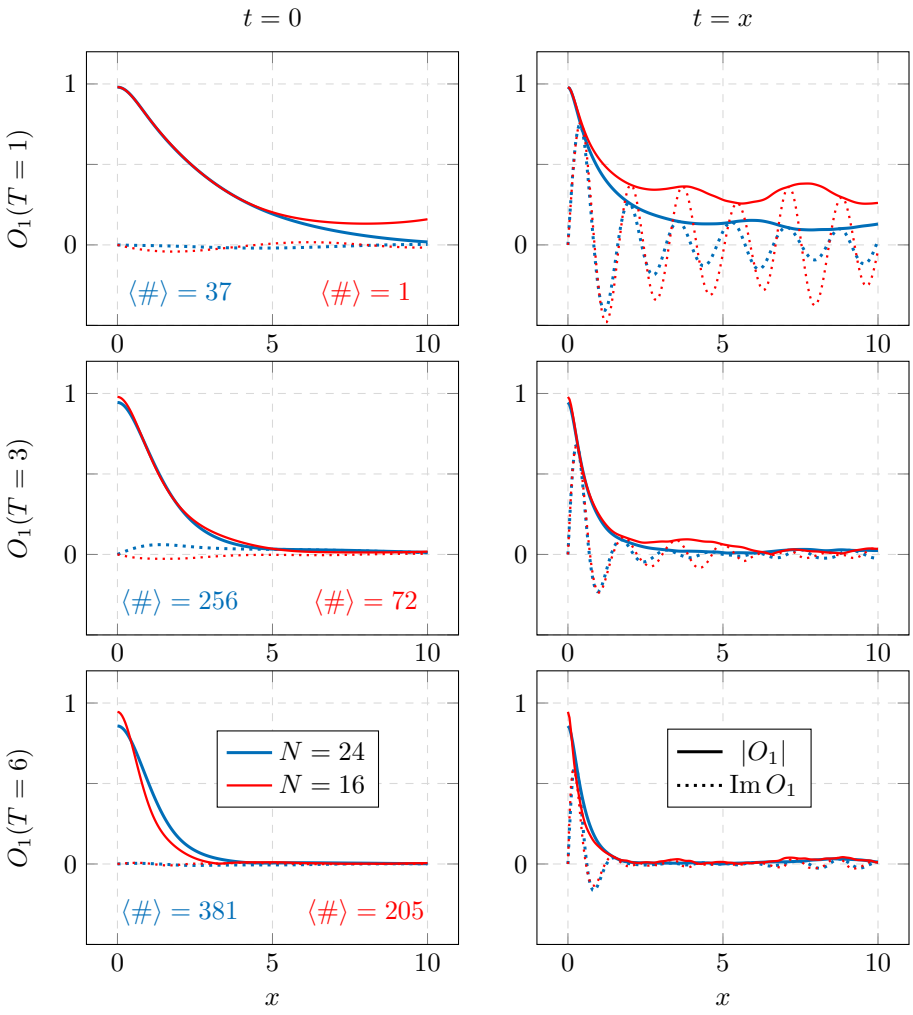


Figure 2.6: Numerical $O_1 := \langle \Psi^\dagger(0,0)\Psi(x,t) \rangle$ via the algorithm, plotted against position x. Blue lines have $N = L = 24$, red $N = L = 16$. Downward, we increase $T \in \{1, 3, 6\}$. The left column is the static case, right the ray $v = 1$. Box size used is $B = 4$, interaction parameter $c = 4$. Each numerical plot is a flat average over 50 sampled bra states. The average number of ket states $\langle \# \rangle$ visited per bra, in units of 10^3, is printed on the left bottom.

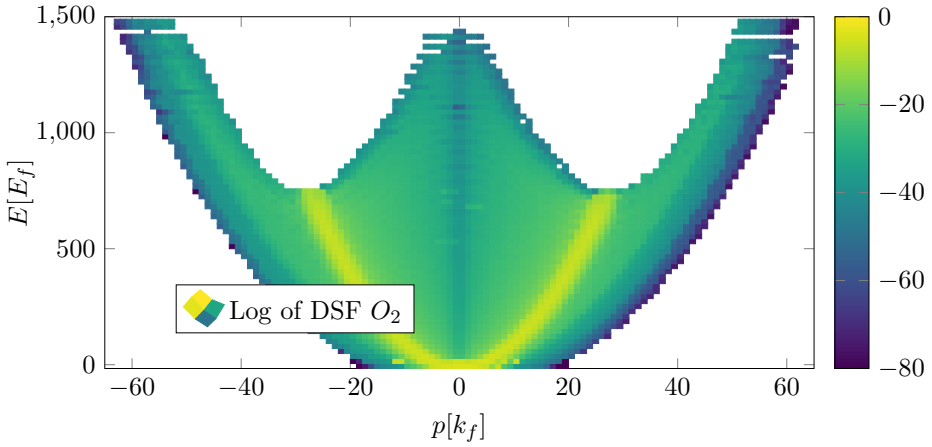
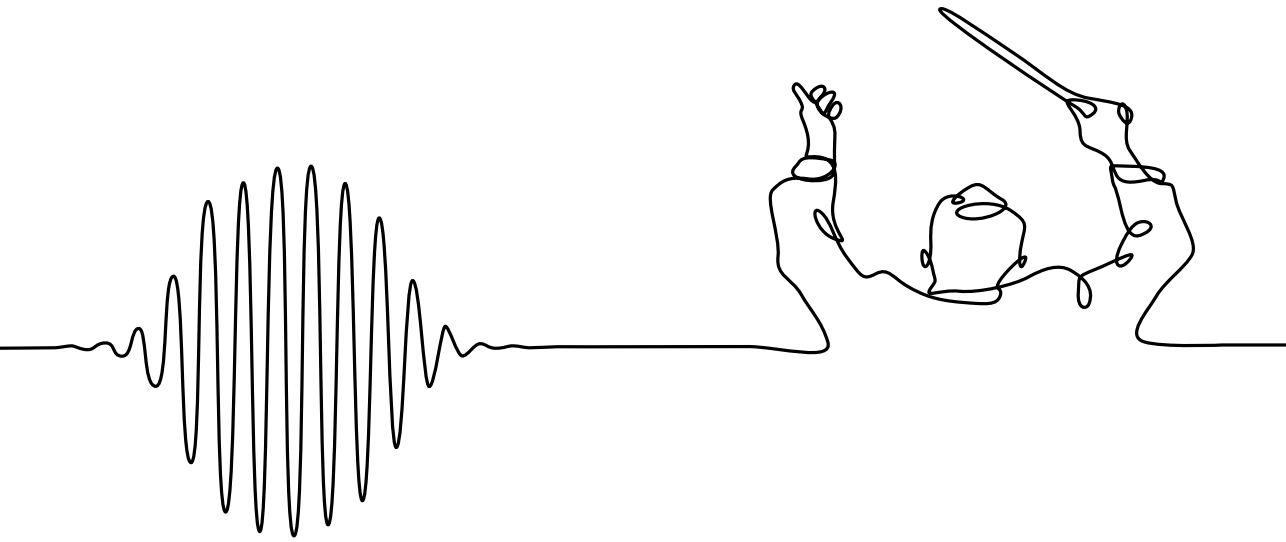


Figure 2.7: Numerical calculation of the log of the Density Structure Factor $O_2 := \langle \Psi^\dagger \Psi \Psi^\dagger \Psi \rangle$ via the algorithm, as a function of the momentum and energy of the contributing states, in units of k_F and E_F. The color corresponds to the log of the weight density in each rectangular bin. $T = 6$, as temperature rises, the figure stretches to higher E and p, but remains qualitatively similar. $N = L = 15$. The heat map shown is a flat average, bin-wise, over that of 50 sampled states, before taking the log.

Part I

Appendices



Appendix A

Subroutines of Hilbert Space Search Algorithm

In this appendix, we briefly explain some of the routines used repeatedly in the main algorithm of section 2.2. They are all available in the 'FFstate.py' file at `https://github.com/DMChernowitz/Free-Fermionic-Hilbert-Search`.

Because it is a recurring theme, it is worthwhile to comment on how all elements of an arbitrarily large tensor product space, such as (2.8), are produced algorithmically, without using excessive memory. The application is any kind of permutation or combinatoric set, see for example A.2, A.3, and A.8. We have made eager use of recursion by *generator* objects. In Python, one can create an object of the generator class. This object is a function that will sit in memory, with its local variable space saved, without running to completion. Every time it is prompted or called, for instance to produce a local shuffling, it runs its code until it yields another output, then halts again. This is preferable, when many combinatoric values must be found, to storing all such results in a list, as each is often only needed once and only one at a time. When the code of the generator terminates, the object is cleared from memory. Once can loop over generators in a similar fashion as over lists. One difference: though the number of outputs produced is deterministic, it may not be known a priori.

The next ingredient is recursion. Such generators may call themselves with a smaller input, tensoring a simple space to a larger space with an already tensored structure, itself the product of recursion. The process starts with the trivial space, and tensors on the composite elements, one by one. This process is evocative of mathematical induction.

A.1 Local Box Shuffling

Name:	**ind_draw**
Purpose:	Provide all local box shufflings for a box allowing the integers $[\mathbf{o}, \mathbf{o}+1, \ldots, \mathbf{m}-1]$, filled with $\mathbf{n}$ quantum numbers.
Inputs:	Lower bounding integer $\geq \mathbf{o}$, upper bounding integer $< \mathbf{m}$, sample size $\mathbf{n}$.
Outputs:	Each call to an instance of **ind_draw** yields a unique ordered list of $\mathbf{n}$ integers between from $[\mathbf{o}, \mathbf{m}-1]$. Terminates after $\binom{\mathbf{m}-\mathbf{o}}{\mathbf{n}}$ outputs.

ind_draw is a generator that calls itself recursively. All shufflings at size $\mathbf{n}$ can be obtained by choosing the first of the $\mathbf{n}$ elements (quantum numbers) to be $k \in [\mathbf{o}, \mathbf{m}-\mathbf{n}]$, and joining it with the all results of a box shuffling from a smaller box: using $k+1$ as lower bounding integer, and $\mathbf{n}-1$ filling. We only take values of k that allow for a legal filling in lower recursion. In order for recursion to terminate, at $\mathbf{n}=0$, the trivial empty list must be returned. As a fail-safe, the algorithm only yields any results for nonnegative $\mathbf{n}$, otherwise recursion would run away to $\mathbf{n} \to -\infty$.

```
1  def ind_draw(o,m,n):
2      if n>=0:
3          if n == 0:
4              yield []
5          else:
6              for k in range(o,m-n+1):
7                  for sub in ind_draw(k+1,m,n-1):
8                      yield [k]+sub
```

A.2 Constrained particle placement

Name:	**boxicles**
Purpose:	Provide all ways to place **n** particles divided over a number of positions, where each position has a maximum allowed occupation.
Inputs:	Number of particles **n**, and a list of nonnegative integers of allowed occupation per position, **depths**.
Outputs:	Each call to an instance of **boxicles** yields a unique list of the same length as the input **depths**, with nonnegative inputs summing to **n**, and never is the k^{th} element of such an output larger than the k^{th} element of **depths**. All possibilities are yielded, filling the positions from left to right, then the routine terminates.

boxicles is a generator that calls itself recursively. All placement profiles with **n** particles can be obtained by adding a single particle to a placement profile of $\mathbf{n}-1$ particles. In order to avoid duplicates, we never add to lower-level profiles at a position that is to the right of any already placed particle. This means we loop over the position k of the profile from left to right, placing an additional particle there if allowed. When we encounter a non-empty position, we may add one particle, then break out of loop. In this manner we achieve a definite ordering of the placement of particles, meaning a profile cannot be produced in multiple ways.

At the lowest level of recursion, $\mathbf{n} = 0$, we must terminate by yielding a list of zeros.

```
1  def boxicles(n,depths):
2      M = len(depths)
3      if n == 0:
4          yield [0 for _ in range(M)]
5      else:
6          for preput in boxicles(n-1,depths):
7              for k in range(M):
8                  if preput[k]<depths[k]:
9                      yield [preput[a]+int(a==k) for a in range(M)]
10                 if preput[k]:
11                     break
```

A.3 All Global Microshufflings at Original Box Filling

Name:	**FFstate.in_box_shuffle**
Purpose:	Provide all global microshufflings corresponding to a certain list of box fillings.
Inputs:	A list **Bfil** of nonnegative integers describing the occupation of each box, $\{\nu_a\}$.
Outputs:	Each call to an instance of **in_box_shuffle** yields 4 variables: 1. A unique microshuffling, described by a list of integers. This is chosen such that an amount of particles are placed in the a^{th} box equal to the a^{th} entry of **Bfil**. 2. the scaled momentum of this shuffling (sum of all integers), **p0** + **p**. 3. the scaled energy of this shuffling (sum of all squared integers), **e0** + **e**. 4. A list **indis** + **d**, equal in length to **Bfil**, of the indices $\{d\}$, corresponding to an enumeration of the local shuffling of each box. After all microshufflings are produced, the routine terminates.

in_box_shuffle is both a method of the **FFstate** class as well as a generator that calls itself recursively. The recursive nature is clear: if we can generate all microshufflings for all boxes up to number $a-1$, extending each of those microshufflings with each local box shuffling of an additional box in turn, constitutes all microshufflings of a total boxes. At the lowest level of recursion, $a = 0$, we must return the empty list to halt. During initialization of the **FFstate** class instance, the scaled momenta and energy of the bra state are stored as **self.pbra** and **self.ebra**. In (1.9) we see that these quantities are taken relative between bra and ket, so at $a = 0$, we begin counting momentum and energy from there, and these are also built up recursively.

Another variable calculated at initialization is **self.U**. It is a deeply nested list of lists. The first index ξ_a denotes the excitation, we may have $\xi_a < 0$ for holes, as Python indexing wraps around when negative. With first index 0 in **self.U**, this refers to boxes with original filling $\{\nu_a\}$. The second index a enumerates the boxes in the staggered fashion described earlier in (2.5). The contents of **self.U**$[\xi_a][a]$ is a list of tuples, each tuple containing 3 elements: a local shufflings of box a, with filling $\nu_a + \xi_a$, along with scaled momentum and scaled energy p, e of that shuffling. For a possible state of **self.U**, see expression (A.1). These local shufflings were all found with **ind_draw**, described in A.1.

By combining recursively all such local box shufflings from **self.U**, all global mi-

croshufflings are produced. In order to know at what multi-index of the **weightrix** (2.20) to put this state, the local shufflings are enumerated with local index d_a (the order in which they appear in **self.U**), and appended to the indices of boxes from lower levels of recursion to form a unique microshuffling multi-index.

```
    def in_box_shuffle(self,Bfil):
        a = len(Bfil)
        if a == 0:
            yield [],self.pbra,self.ebra,[]
        else:
            for leftside,p0,e0,indis in self.in_box_shuffle(Bfil[:a-1]):
                d=0
                for nowbox,p,e in self.U[0][a-1]:
                    yield leftside + nowbox, p0+p,e0+e,indis+[d]
                    d += 1
```

A.4 Integrate Out All But One Tensor Dimension

Name:	**dissect**
Purpose:	Take a multidimensional tensor (Numpy Array), and return a list, for each dimension, of that axis with all other dimensions integrated out.
Inputs:	Multidimensional array of floating point numbers, **We**.
Outputs:	List of one-dimensional Numpy arrays **w**, whose first index enumerates the dimensions/ axes of **We**, and whose second index corresponds to the position along that axis in **We**. The value of $\mathbf{w}[a][d] = w_{d_a}(a)$ from (2.20).

dissect takes as argument what is termed the **weightrix** in the main algorithm, at the stage of producing all microshufflings at original box fillings. It returns the proxy weights for each box/dimension, with all other boxes/dimensions summed and normalized out. **A** holds the size of each dimension of **We** in a list. We loop over the **M** inner boxes with the following protocol:

- If a box **a** only has the trivial filling ($\mathbf{A}[\mathbf{a}] = 1$), there is no need to calculate the normalized proxy weight of the corresponding axis, as it is by definition unit.
- If, conversely, box **a** is nontrivial, containing multiple distinct local shufflings, we must sum over all other axes and add this axis to the output.
- We modify **We**, by summing over its current first axis, collapsing the array to a smaller dimension. This axis has been extracted and subsequent axes will only need this axis summed.

Finally we normalize each box by the total weight collected, so each axis sums to unity, as defined in (2.20).

```
def dissect(We):
    w = []
    A = We.shape
    M = len(A)
    for a in range(M):
        if A[a] >1:
            w.append(We.sum(tuple(range(1,M-a))))
        else:
            w.append(arr([1]))
        We = We.sum(0)
    for a in range(M):
        if A[a] >1:
            w[a] /= We
    return w
```

A.5 Divide Local Shufflings into Tiers

Name:	**FFstate.sortout**
Purpose:	Divide extremal local shufflings at a given level λ and pad γ into tiers $\omega_a(z)$, according to proxy weights.
Inputs:	list of Numpy arrays **odw**, each array corresponding to box a, the arrays holding proxy weights $w_{d_a}(a)$. Level **nl** and **pad** determine where to store the result, as well as which boxes are involved.
Outputs:	None. The effect is an updated **self.BT** variable.

sortout is a method of the **FFstate** class that updates the **FFstate** variable **self.BT**. The latter is a deeply nested list of lists that holds the box tiers $\omega_a(z)$ of the bra state instance. The first index denotes the excitation, $\xi_a =$ **nl** and may be zero, for the original filling ν_a of the box, or a positive or negative integer, denoting that many particles or holes in the box, respectively. The second index of **self.BT** is the box index a. If we are at **pad** $= 0$, this routine treats all M inner boxes, those originally filled in the bra state. If **pad** > 0, the routine only treats the outer two values for a. The third index of **self.BT** is the local tier ι_a. For $\iota_a = 0$, in **self.BT** we find a list of tuples, copied over from **self.U**, corresponding to local box shufflings that have the expected highest weight for this given box and filling $\nu_a + \xi_a$. Higher tiers $\iota_a > 0$ may or may not exist, and are populated by local shufflings at least a factor **self.eps** smaller in proxy weight than the previous tier, see equation (2.13) for an example. Proxy weights **odw** have be calculated as in (2.20) if (**pad** $= 0$, **nl** $= 0$), and else by (2.21).

In order to achieve this structure, we find the order of the descending proxy weights $w_{d_a}(a)$. Looping over d_a in that order, as long as $w_{d_a}(a) > \epsilon^{b+1}$, we append shuffling

d_a from **self.U** to tier b in **bt**. If not, we increment b by one. After the loop, **bt** $(\omega_a(\nu_a + \xi_a))$ becomes a new element of **self.BT**, which henceforth be queried at higher λ or γ.

The log bin size, $\epsilon =$ **self.eps** is set to 0.3 by default, but can be chosen as an input to the **FFstate** instance.

```
    def sortout(self,odw,nl,pad):
        if pad:
            a = self.innerboxes+2*(pad-1)
        else:
            a = 0
        for w in odw:
            argord = (-w).argsort()
            noweps = 1
            bt = []
            for j in argord:
                if not (w[j] > noweps):
                    noweps *= self.eps
                    bt.append([])
                bt[-1].append(self.U[nl][a][j])
            self.BT[nl][a] = bt
            a += 1
```

A.6 Generate All States in a Block

Name:	**FFstate.deepen_gen**
Purpose:	Object left in memory, can be prompted to produce volleys, small subsets of states from the block defined by the inputs, and update the operator τ and the sum rule with them. These states are disjunct from those produced in the **explore** (A.10) phase.
Inputs:	**level**, **pad**, and **tier** (λ, γ, ι) of the block under consideration.
Outputs:	The total weight **running** found in each volley, and the number of states **count** in the volley.

deepen_gen is a method of the **FFstate** class that, when called for the next yield, considers a volley of all states corresponding to a single excitation profile $\{\xi_a\}$ from the block, with (almost) all local tier placements $\{\iota_a\}$ such that $\sum_a \iota_a = \iota$, and returns statistics on the volley. During the explore phase, the states generated have $\iota_a = 0$ on non-extremal boxes, and all possible local shufflings, thus all possible tiers, on extremal boxes. In order to have different states at this phase of the search, at least one non-extremal box a must have $\iota_a > 0$. In other words, for extremal boxes $\tilde{a}$, we cannot have $\sum_{\tilde{a}} \iota_{\tilde{a}} = \iota$.

Each yield is calculated from a variable **PES**: a list, in turn stored listed in the **FFstate** variable **p_h_profiles**, at first index λ, second index γ. The first entry of

PES is an excitation profile $\{\xi_a\}$ in the style of (2.11). The second is a list, length between zero and two, of the indices $\tilde{a}$ of extremal boxes. The third is the **walking** variable, a total amount of overlap found at this profile, including phase. This is useful for other investigations into the nature of field theory macrostates. Given the profile $\{\xi_a\}$ **PES**[0] and extremal boxes **PES**[1], we can produce all possible placements of the local tiers to achieve **tier** global tier ι: the method **tier_profile**, (see A.7), distributes the global tier in all ways over the available boxes in the profile such that the maximum local tier is never exceeded, and that not all nonzero local tiers are found in the extremal boxes. With each $\{\iota_a\}$ **tierpf**, we may produce all possible $|\mathbf{k}\rangle$ states **ket** and corresponding momentum and energy **p**, **e**, with the method **shuf_prof_tier**, from A.8. Inside this loop, we call the **FFstate** method **update_tau**, seen in A.9, to update the τ-function and the sum rule. After the volley is done, the **walking** variable is incremented, and the **running** (sum of weight w in the volley) and **count** (number of states in the volley) are returned as a criterion for the success of this volley. $\langle w \rangle$ from **self.cart** in (2.22) is their quotient. The more weight returned in the fewer states, the better.

```
    def deepen_gen(self,level,pad,tier):
        for PES in self.p_h_profiles[level][pad]:
            walking = 0
            running = 0
            count = 0
            for tierpf in self.tier_profile(PES[0],PES[1],tier):
                for ket,p,e in self.shuf_prof_tier(PES[0],tierpf):
                    count += 1
                    w,w2 = self.update_tau(ket,p,e)
                    running += w2
                    walking += w
            PES[2] += walking
            yield running,count
```

A.7 Distribute Local Tiers over Boxes

Name:	**FFstate.tier_profile**
Purpose:	Provide all distributions of local tiers $\{\iota_a\}$ over boxes that sum to a given global tier ι.
Inputs:	The excitation **profile** $\{\xi_a\}$, in the format of equation (2.11), a list of extremal box indices $\tilde{a}$ **EJ**, and the global **tier** $= \iota - 1$ needed on this profile.
Outputs:	Each call to an instance of **tier_profile** returns a list $\{\iota_a\}$ of integers, **tierpf**, each entry of which corresponds to a box a in **profile**, and each value of which is the local tier of local shufflings to be concatenated to global states by the function that calls **tier_profile**.

tier_profile is both a method of the **FFstate** class as well as a generator. Its first action is to tabulate in **deps**, from **self.BT**, how many 'excited' local tiers $\{\#\omega_a(\nu_a + \xi_a)\}$ are available for each box at present filling in the input **profile**[1]. Then **boxicles** can place the local tiers in all permutations (see A.2). Rather than call it directly, we must ensure that not all nonzero local tiers end up in extremal boxes, as those states were considered during **explore**. If there is a single extremal box, or one entry to **EJ**, we can achieve this by artificially lowering (if necessary) the allowed local tier in that box to below the global tier ι, then naturally the remainder spills over into other boxes. If instead, there are two extremal boxes and two entries to **EJ**, we must distribute the $\{\iota_a\}$ with more care. We determine the total capacity of the extremal boxes, and iterate over placing between none and all but one of the local tiers on them together, by temporarily treating them as a single box. Inside one such iteration, we redistribute the local tiers between the two extremal boxes in all ways manually. Finally, if there are no extremal boxes, straightforward calling of **boxicles**, suffices.

At least one local tier must be placed, if we are to have different states than the exploration phase. Nomenclature is therefore such that the minimal value of **tier** in **tier_profile** is zero, so **boxicles** is called with $\iota - 1 =$ **tier** excitations so as to not repeat the exploration states.

[1] If the only tier available in box a is $\iota_a = 0$, we cannot place excited tiers here, so **deps**[a]=0.

```
    def tier_profile(self,profile,EJ,tier):
        deps = [len(self.BT[profile[j]][j])-1
                for j in range(len(profile))]
        if len(EJ)==1:
            deps[EJ[0]] = min(deps[EJ[0]],tier)
        if len(EJ)==2:
            da,db = deps[EJ[0]],deps[EJ[1]]
            deps[EJ[0]] = min(da+db,tier)
            deps[EJ[1]] = 0
            for tierpf in boxicles(tier+1,deps):
                if tierpf[EJ[0]] > da:
                    tierpf[EJ[1]] = tierpf[EJ[0]] - da
                    tierpf[EJ[0]] = da
                yield tierpf
                while tierpf[EJ[1]] < db and tierpf[EJ[0]] > 0:
                    tierpf[EJ[1]] += 1
                    tierpf[EJ[0]] -= 1
                    yield  tierpf
        else:
            for tierpf in boxicles(tier+1,deps):
                yield tierpf
```

A.8 All Microshufflings at Excitation and Tier Profile

Name:	**FFstate.shuf_prof_tier**
Purpose:	Provide all microshufflings at a given excitation profile and tier profile.
Inputs:	Two lists: the excitation profile $\{\xi_a\}$, **phpf** in the format of equation (2.11), and a tier profile $\{\nu_a\}$, **tierpf**, consisting of nonnegative integers.
Outputs:	Each call to an instance of **shuf_prof_tier** returns a tuple of three terms: a microshuffling concatenating **tail**$+$**locshuf**, the microshuffling's scaled momentum **p**$+$**p0**, and the microshuffling's scaled energy **e**$+$**e0**.

shuf_prof_tier is both a method of the **FFstate** class as well as a generator that calls itself recursively. It builds the microshufflings from the left. At the deepest level of recursion, both profiles are length 0, and we return the empty shuffling, and the gauge choice of momentum and energy **self.pbra** and **self.ebra**. On top of this, for each possible left part of the microshuffling, **tail**, found by calling **shuf_prof_tier** with curtailed profiles, we return sequentially all local shufflings from **self.BT** at the correct particle-hole-excitation ξ_a, **phpf**$[-1]$ and the correct local tier ι_a, **tierpf**$[-1]$

for the next box to the right, along with that box's momentum and energy. The outermost layer of this function produces all full length microshufflings.

```
    def shuf_prof_tier(self,phpf,tierpf):
        l = len(phpf)
        if l==0:
            yield [],self.pbra,self.ebra
        else:
            for tail,p,e in self.shuf_prof_tier(phpf[:-1],tierpf[:-1]):
                for locshuf,p0,e0 in self.BT[phpf[-1]][l-1][tierpf[-1]]:
                    yield tail+locshuf,p+p0,e+e0
```

A.9 Update τ-function

Name:	**FFstate.update_tau**
Purpose:	Take a set of momentum quantum numbers, calculate the overlap to the bra state $\langle \mathbf{g}|$ used in the initialization, update the progress of the τ-function and sum rule.
Inputs:	Quantum numbers **ket**, and corresponding sum, or total scaled momentum **p**, and sum of squares or total scaled energy, **e**.
Outputs:	The overlap $\langle g|k\rangle$, **w** and its squared modulus, the weight **w2** of the state.

This subroutine also updates **FFstate** variables **self.tau**, **self.sum_rule** and **self.states_done**, which should be self-explanatory after chapter 2.1.

```
    def update_tau(self,ket,p,e):
        detti = det(1/(self.abra-sorted(ket)-self.phi))
        w = detti*self.presin
        w2 = w*w
        self.tau += w2*exp(p*self.Place+e*self.Time)
        self.sum_rule += w2
        self.states_done +=1
        return w,w2
```

A.10 Explore Hilbert Space Column

Name:	**FFstate.explore**
Purpose:	Create excitation profiles and local tiers needed to produce all states in a column of blocks, given by λ, γ, and produce the first of these states in the process.
Inputs:	The λ, **level** and γ, **pad** to be treated.
Outputs:	None. The effect is an updated **self.BT** variable with local tiers $\omega_a(\nu_a \pm \lambda)$, updated **self.p_h_profiles** variable with excitation profiles $\{\xi_a\}$, an updated **self.blocks** and **self.cart** variable reflecting the new available blocks to query.

explore is a method of the **FFstate** class that performs an important function in the Hilbert space search, by preparing all objects needed to produce all states in the blocks over a given pad-level combination. If the level is zero, there are no excitations and the pads can have no particles. There are no new states here after **ground_shuffle**. The script only needs to expand the size of **FFstate** objects **self.BT** and **self.blocks** to allow the algorithm to call these objects at larger pad indices, although they return empty lists as fillings.

However, if the level is nonzero, new states must be produced. The algorithm prepares some auxiliary variables, such as the number of boxes to be considered **b0** and variables of the **FFstate** class **self.spind** and **self.spox**. These variables are updated by the closely related **FFstate** method **parasite**, described in A.11. **self.spind** has two entries, the first for particles and the second for holes, and each keeps track of the index $d_{\tilde{a}}$ of the enumerated local shufflings of an extremal box, if one is present. This means, as we build up global microshufflings recursively, at the depth where we iterate over an extremal box, **self.spind**[0] holds the integer counting inside **self.U**$[\xi_{\tilde{a}}][\tilde{a}]$ which local shuffling is currently being used in the iteration, for the extremal box with $\xi_{\tilde{a}} = \lambda$. **self.spind**[1] does the same for $\xi_{\tilde{a}} = -\lambda$. **self.spox** on the other hand holds the box index $\tilde{a}$ of the extremal box, again the first entry for particles, the second for holes. These data are useful in order to put the proxy weight in the correct location in the **weightrix** that is filled during the course of **explore**. **self.spox** and **self.spind** are updated by the method **parasite**, discussed later on. Recall that extremal boxes are the ones for which the local tiers $\omega_{\tilde{a}}(\nu_a \pm \lambda)$ still need to be constructed, as in example (2.12).

The next division is whether $\gamma > 0$. If yes, then the only extremal boxes can be the outer two, and only with particles, as there can be no holes in empty boxes. Moreover, if $\lambda < B =$ **self.B**, then there are no extremal boxes because the largest box excitations have already been treated at $\lambda = B$. So for nonzero pad, $\lambda \leq B$, we create a **weightrix** with two indices for the two outer boxes, and in each of those, $\binom{B}{\lambda}$ zeros to hold the proxy weight, one for each local shuffling. See also equation (2.21). Then for the boxes introduced in the new pad, we must add new entries to variables holding local shufflings. The first is the **FFstate** variable **self.U**, whose first index is ξ_a, whose second is a, and which must now be extended in this second dimension.

The content of an entry of **self.U** at a given level and box is a list of tuples. Such a tuple contains a local shuffling, its scaled momentum and scaled energy. From the example of the main text, (2.3), $B = 5$ and $\nu_0 = 3, \nu_3 = 1$, we can illustrate some of the elements[2] of

self.U$[\xi_a][a] =$

ξ_a	$a = 0$	...	$a = 3$	...
0	$\begin{bmatrix}[[0,1,2],3,5],\\ \dots \\ [[2,3,4],9,29]\end{bmatrix}$	...	$\begin{bmatrix}[[-10],-10,100],\\ \dots \\ [[-6],-6,36]\end{bmatrix}$	...
1	$\begin{bmatrix}[[0,1,2,3],6,14],\\ \dots \\ [[1,2,3,4],10,30]\end{bmatrix}$	...	$\begin{bmatrix}[[-10,-9],-19,181],\\ \dots \\ [[-7,-6],-13,85]\end{bmatrix}$	...
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$
-2	$\begin{bmatrix}[[0],0,0],\\ \dots \\ [[4],4,16]\end{bmatrix}$	...	$\begin{bmatrix}\\ \\ \end{bmatrix}$	...
-1	$\begin{bmatrix}[[0,1],1,1],\\ \dots \\ [[3,4],7,25]\end{bmatrix}$	...	$\begin{bmatrix}[[],0,0]\\ \\ \end{bmatrix}$	...

(A.1)

This is built up level by level and pad by pad during exploration. The entries themselves are created by the method **local_shuffle**, which essentially applies **ind_draw** (see A.1) and sums its piece-wise first and second power.

The next variable to be updated is **self.BT**, which holds the $\omega_a(z)$. This comprises the same elements as **self.U**, except divided one index further, after $[\xi_a][a]$ into tiers $[\iota_a]$, and fourth index enumerating the local shufflings. **self.BT** also introduces our companion function **parasite**, from section A.11. **self.BT** is a kind of database, and is queried by combinatoric methods such as **shuf_prof_tier** (see A.8). However, at present time, the tiers are not known for the extremal boxes. The solution is to have the extremal boxes, when they arise in the excitation profiles, behave as if they only have the lowest tier, zero, and produce all states with global tier zero. This includes all possible local shufflings in extremal boxes, and only the actual $\iota_a = 0$ shufflings in other boxes. During, we wish to keep track of which local shuffling is used in the extremal boxes, in order to add the weights of these states to the entry of the **weightrix** that corresponds to said shuffling. Instead of a simple list, we install a tailor-made generator, **parasite**, at the appropriate location in **self.BT**. In Python, entering negative indices loops around to the end of the list: for a list **my_list** of length **l**, **my_list[l-a]** = **my_list[-a]**. For this reason, the structure of the first index of **self.U** and **self.BT** is {no excitation, 1 particle, 2 particles, ..., 2 holes, 1 hole}. As we wish to query **self.BT** for larger numbers of holes and particles, we

[2]When $\xi_a + \nu_a$ does not lie in $[0, \dots, B]$, there are no legal local fillings and the element of **self.U** is an empty list.

must take care to insert the higher excitations in the middle of the first index, the outer list.

The following loops keep track of how many particles are in the outer pads, at least one and at most the level, or $2B$, then how these particles are divided between left and right, and then how the rest of the particles and holes distribute among the remaining boxes, by means of **boxicles** (A.2). First particles are placed, then the holes are allowed in the remaining free spots. This produces all possible excitation profiles at this level and pad, which are stored, and per profile states are produced as described above and their weights entered into the weightrix. The **FFstate** variable **p_h_profiles** is extended in order of decreasing state-averaged weight found in the profiles, the necessary tiers are created with **sortout** (A.5), replacing the **parasite** instances, and **self.blocks** is updated with the generator method **deepen_gen**, from A.6.

Conversely, if the pad is zero, the **weightrix** has an extra first index to allow for extremal boxes with holes, and we do not need to artificially place any number of particles in the outer boxes. Besides that, much is the same.

A necessary check is that the created block has any actual states at the lowest untreated global tier, which is $\iota = 1$, at **self.blocks**[**level**][**pad**][0]. If so, the block is added to the **self.cart** and may be prompted for more states in the future. One final caveat: empirically it is desirable to suppress newly minted blocks, which are on the periphery, from being searched directly. When new blocks become the current searched block, even more are explored, which is computationally expensive. For this reason, the results of **explore** are initially added to the cart with an average weight that is scaled by a constant, called **self.carteps**. Testing has indicated that the most efficient search is performed with a default constant **self.carteps** = 0.2, but this constant can be chosen as an input to the **FFstate** instance.

```
    def explore(self,level,pad):
        if level:
            b0 = self.innerboxes+2*pad
            flat = [0 for _ in range(b0)]
            self.spind = [None,None]
            self.spox = [None,None]
            if len(self.blocks[level-1]) <= pad:
                self.explore(level-1,pad)
            if pad:
                if level <= self.B:
                    partconf = ncr(self.B,level)
                    z0 = [0 for _ in range(partconf)]
                    weightrix = [z0 for _ in [0,0]]
                    for boxj in range(b0-2,b0):
                        gl = self.local_shuffle(level,self.box2qn(boxj))
                        self.U[level].append(gl)
                        self.BT[level].append([self.parasite(level,boxj)])
                profnweight = []
```

```
                for oparts in range(1,min(level,2*self.B)+1):
                    ma,mi = max(0,oparts-self.B),min(oparts,self.B)
                    for opl in range(ma,mi+1):
                        opr = oparts-opl
                        de = self.boxemp+[self.B for _ in range(2*pad-2)]
                        for partplace in boxicles(level-oparts,de):
                            holesdeps = []
                            for j in range(self.innerboxes):
                                if partplace[j]:
                                    holesdeps.append(0)
                                else:
                                    holesdeps.append(self.boxfil[j])
                            for holeplace in boxicles(level,holesdeps):
                                prof = partplace+[opl,opr]
                                running = 0
                                walking = 0
                                ct = 0
                                for jj in range(self.innerboxes):
                                    prof[jj] -= holeplace[jj]
                                spt = self.shuf_prof_tier(prof,flat)
                                for ket,p,e in spt:
                                    w,w2 = self.update_tau(ket,p,e)
                                    running += w2
                                    walking += w
                                    ct += 1
                                    if self.spox[0] != None:
                                        i1,i2 = self.spox[0]%2,self.spind[0]
                                        weightrix[i1][i2] += w2
                                if self.spox[0] == None:
                                    extholes = []
                                else:
                                    extholes = [self.spox[0]]
                                    self.spox[0]=None
                                bu,mc = [prof,extholes,walking],max(1,ct)
                                profnweight.append([running/mc,bu])
                np = [xp for _,xp in sorted(profnweight,reverse=True)]
                self.p_h_profiles[level].append(np)
                if level <= self.B:
                    odw = []
                    for jw in [0,1]:
                        sw = sum(weightrix[jw])
                        odw.append(arr(weightrix[jw])/sw)
                    self.sortout(odw,level,pad)
                self.blocks[level].append([self.deepen_gen(level,pad,0)])
            else:
                profnweight = []
                if level <= self.B:
```

```
                    weightrix = []
                    for nl in [-level,level]:
                        weightrix.append([])
                        for bb in self.boxfil:
                            z0 = [0 for _ in range(ncr(self.B,bb-nl))]
                            weightrix[-1].append(z0)
                        nou = []
                        self.BT.insert(level,[[self.parasite(nl,boj)]
                        for boj in range(self.innerboxes)])
                        s=0
                        for boxj in range(self.innerboxes):
                            nou.append(self.local_shuffle(
                                self.boxfil[boxj]+nl,s))
                            if s<0:
                                s = -s
                            else:
                                s = -s-self.B
                        self.U.insert(level,nou)
                for partplace in boxicles(level,self.boxemp):
                    holesdeps = []
                    for j in range(self.innerboxes):
                        if partplace[j]:
                            holesdeps.append(0)
                        else:
                            holesdeps.append(self.boxfil[j])
                    for holeplace in boxicles(level,holesdeps):
                        prof = [partplace[jj]-holeplace[jj]
                        for jj in range(self.innerboxes)]
                        running = 0
                        walking = 0
                        ct = 0
                        for ket,p,e in self.shuf_prof_tier(prof,flat):
                            w,w2 = self.update_tau(ket,p,e)
                            running += w2
                            walking += w
                            ct += 1
                            for hw in [0,1]:
                                if self.spox[hw] != None:
                                i1,i2 = self.spox[hw],self.spind[hw]
                                    weightrix[hw][i1][i2] += w2
                        extholes = []
                        for sw in [0,1]:
                            if self.spox[sw]!= None:
                                extholes.append(self.spox[sw])
                                self.spox[sw] = None
                        bu,mc = [prof,extholes,walking],max(1,ct)
                        profnweight.append([running/mc,bu])
```

```
                np = [xp for _,xp in sorted(profnweight,reverse=True)]
                self.p_h_profiles.append([np])
                if level <= self.B:
                    for jw in [0,1]:
                        odw = []
                        for jww in range(self.innerboxes):
                            sw = sum(weightrix[jw][jww])
                            odw.append(arr(weightrix[jw][jww])/sw)
                        self.sortout(odw,[level,-level][jw],0)
                self.blocks.append([[self.deepen_gen(level,0,0)]])
            for rm,cm in self.blocks[level][pad][0]:
                if cm:
                    self.cart.append([rm/cm*self.carteps,[level,pad,0]])
                    break
        else:
            self.BT[0].extend([[[[[],0,0]]],[[[],0,0]]])
            self.blocks[0].append([[]])
```

A.11 Placeholder Tier Zero

Name:	**FFstate.parasite**
Purpose:	Employed by **explore**, from A.10. Takes the place of the tier-zero list of local shufflings of an extremal box, inside **self.BT**, and return all local shufflings while updating auxiliary variables used to fill the **weightrix**.
Inputs:	The excitation ξ_a, **nl**, positive for particles and negative for holes, and the box index a at which it is located, **boxj**.
Outputs:	Each call to an instance of **parasite** yields a local box filling as stored in **self.U**.

An instance of **parasite** is created, and located at **self.BT**[**nl**][**boxj**][0], by **explore**, while $\pm\lambda = \xi_a =$ **nl**, and **boxj** is the index a of an extremal box. When **explore** is using **shuf_prof_tier** from A.8 to concatenate local shufflings with $\iota = 0$ recursively, from time to time it needs those of the unexplored extremal boxes. In that case, instead, **parasite** provides all possible fillings, (as if the extremal box had only tier 0), while storing in the **FFstate** variables **self.spox** the box number of the extremal box, and in **self.spind**, the index of the local shuffling returned. For either particles (**ind** = 0) or holes (**ind** = 1), there is at most a single extremal box at a time. When all microshufflings of an excitation profile have been yielded, which share the same **self.spox**, both these auxiliary variables are reset. This setup is meant to mimic exactly the behavior of the boxes whose tiers are known at other locations in **self.BT**, from the point of view of **shuf_prof_tier**. At the same time, this allows **explore**, running at a higher scope and calling **parasite**, to fill the weights calculated

into the correct locations in the **weightrix** for all local shufflings. As each box is iterated over a variable amount of times, **parasite** restores a new instance of itself at the same location **self.BT[nl][boxj][0]** before stopping the iteration and terminating.

```
    def parasite(self,nl,boxj):
        ind = int(nl<0)
        self.spox[ind]=boxj
        for j,locfil in enumerate(self.U[nl][boxj]):
            self.spind[ind]=j
            yield locfil
        self.BT[nl][boxj][0] = self.parasite(nl,boxj)
```

A.12 Binomial Coefficient

Name:	**ncr**
Purpose:	Calculate the binomial coefficient.
Inputs:	Choose **r** from **n** distinguishable choices.
Outputs:	the amount of ways this can be done, $\binom{\mathbf{n}}{\mathbf{r}}$.

This is an efficient code for binomial coefficients on integers, using falling factorials. It is included because Numpy or the Python Math module do not have it natively, however the algorithm makes use of it.

```
def ncr(n,r):
    if r < 0:
        return 0
    p,q = 1,1
    for j in range(r):
        p *= n-j
        q *= j+1
    return p//q
```

Bibliography

[1] D. Chernowitz and O. Gamayun, "On the Dynamics of Free-Fermionic Tau-Functions at Finite Temperature," *SciPost Phys. Core*, vol. 5, p. 6, 2022.

[2] S. Viefers, P. Koskinen, P. Singha Deo, and M. Manninen, "Quantum rings for beginners: Energy spectra and persistent currents," *Physica E: Low-dimensional Systems and Nanostructures*, vol. 21, pp. 1–35, 02 2004.

[3] F. M. Cucchietti, D. A. R. Dalvit, J. P. Paz, and W. H. Zurek, "Decoherence and the loschmidt echo," *Phys. Rev. Lett.*, vol. 91, p. 210403, Nov 2003.

[4] K. K. Kozlowski and V. Terras, "Long-time and large-distance asymptotic behavior of the current–current correlators in the non-linear schrödinger model," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2011, p. P09013, sep 2011.

[5] P. Calabrese and J.-S. Caux, "Dynamics of the attractive 1d bose gas: analytical treatment from integrability," *Journal of Statistical Mechanics Theory and Experiment*, vol. 2007, 07 2007.

[6] E. H. Lieb and W. Liniger, "Exact analysis of an interacting bose gas. i. the general solution and the ground state," *Phys. Rev.*, vol. 130, pp. 1605–1616, May 1963.

[7] J.-S. Caux and P. Calabrese, "Dynamical density-density correlations in the one-dimensional bose gas," *Phys. Rev. A*, vol. 74, p. 031605, Sep 2006.

[8] C.-N. Yang and C. P. Yang, "Thermodynamics of one-dimensional system of bosons with repulsive delta function interaction," *J. Math. Phys.*, vol. 10, pp. 1115–1122, 1969.