DiracQ

A Quantum Many-Body Physics Package *

John G. Wright and B. Sriram Shastry

Physics Department

University of California Santa Cruz, CA 95064

September 7, 2015

Website of **DiracQ**

`http://github.com/jgwright1986/DiracQ`

Or

`http://physics.ucsc.edu/~sriram/DiracQ`

♦ 1. Introduction:

We present a software package **DiracQ** , for use in quantum many-body Physics. It is designed for helping with typical algebraic manipulations that arise in quantum Condensed Matter Physics and Nuclear Physics problems, and also in some subareas of Chemistry. DiracQ is invoked within a *Mathematica* session, and extends the symbolic capabilities of *Mathematica* by building in standard commutation and anticommutation rules for several objects relevant in many-body Physics. It enables the user to carry out computations such as evaluating the commutators of arbitrary combinations of spin, Bose and Fermi operators defined on a discrete lattice, or the position and momentum operators in the continuum. Some examples from popular systems, such as the Hubbard model, are provided to illustrate the capabilities of the package.

A word about the underlying philosophy of this program is appropriate here. Physicists approach calculations in quantum theory with a certain informality that contrasts with the approach of mathematicians, who usually insist on a somewhat more rigorous (notational) framework. Some formality is also present in the method of programming a computation with older computer languages such as Fortran. The physicist's informality refers specifically to a deferment of definitions and properties of objects, until one actually needs them. This enables one to write down and manipulate and to compound expressions, often into simple and useful final results. In our view the physicist's approach owes much to the notation introduced by Dirac in 1927. In the classic paper "The Physical Interpretation of Quantum Dynamics", P. A. M. Dirac Proc. Roy. Soc.**A113**,621(1927), we find the first mention of c-numbers and q-numbers as follows:

... one cannot suppose the dynamical variables to be ordinary numbers (c-numbers), but may call them numbers of a special type (q-numbers).

This inspired notation helps us to treat the commuting parts of the expression with standard care (due their c-number nature), while reserving extra care for handling the q-number parts.

In **DiracQ** , we initially declare a set of symbols to be operators, i.e. q-numbers. The properties of standard operators such as Bose, Fermi, position-momentum, angular momentum etc are already programmed, and if required, a user could define more exotic operators with specific properties. Once this is done, any expression is split into its c-number and q-number parts as

the first and fundamental operation. Complicated input expressions may be written with a fairly informal syntax, with mixtures of c-numbers and q-numbers, and as the sums of such expressions. **DiracQ** handles them by first separating them into their c-number and q-number parts. Standard operations on operators, such as adding, multiplying or commuting them is then straightforward, the c-number parts are spectators for most part while the q-numbers are combined using the given rules. Finally the expressions are written back in standard input like notation. The package **DiracQ** consists of a set of mutually dependent functions to perform most of the operations. These functions are most often named using the last letter as "Q"; for example we denote a function (described later) as SimplifyQ[a], thus distinguishing it from the function Simplify[a] of *Mathematica* .

This version of **DiracQ** uses commands that call upon the symbolic capabilities of *Mathematica*, and exploits its ability to deal with non commuting symbols. It might be possible to couple the commands of **DiracQ** with an OSS program such as Sage, in view of the relatively small fraction of commands of *Mathematica* that are actually used here. The present version of **DiracQ** was developed and tested on Version 8 of *Mathematica*, and requires the palettes feature of this version in order to declare the standard non commuting symbols. A version avoiding the palettes is planned for the future.

♦ 2. Contents:

The package distribution contains the following files: (X is the version number)

1. **Introduction.pdf**

   This article.

2. **DiracQ_VX.m**

   The main package implementing **DiracQ** .

3. **Getting_Started_and_Tutorial_VX.nb**

   This contains a first introduction to loading **DiracQ** and simple examples, as well as a more detailed overview of the commands and their illustration in simple problems. This notebook has some overlap with the notebook Glossary_VX.nb, it provides an in depth commentary on

the usage of the various functions defined in the package. Simple examples with harmonic oscillator operators are provided as warming up. Examples are provided for all the functions available in **DiracQ** . Users who would like to enlarge the class of operators and define their own set, would find a helpful example treated, where the defining relations of the Virasoro algebra are added, and a few simple computations with their generators carried out.

4. **Glossary_VX.nb**

   A list of all commands in **DiracQ** , their description and a simple example or two of their usage. It also contains, under the heading **DiracQPalette**, a description of the palette used in this notebook, with its various predefined operators and instructions on how to "turn them on".

5. **Examplebook_VX.nb**

   A set of calculations using of **DiracQ** in problems of various level of difficulty. Harmonic oscillator commutations provide a simple introduction to the package, followed by a demonstration of the conservation of the Runge-Lenz-Laplace vector in the Hydrogen atom. The Bra and Ket objects of **DiracQ** are introduced. Fermionic operators are illustrated in the context of the Hubbard model, where the commutators of combinations of Fermi operators with the Hamiltonian are evaluated, leading to exact expressions for the exact second and third moments of the electron spectral function. Applications to spin half particles and Pauli matrices are given using R. J. Baxter's celebrated proof (1971) of the integrability of the 8-vertex model. The relations between the Boltzmann weights required for satisfying the star triangle relations are recovered using **DiracQ** . Also provided is a more complex example based on the 1-d Hubbard model. Its S matrix (Shastry 1986) is known, which can further be used to construct an inhomogeneous generalization of the Hubbard model. This construction is dependent upon the S matrix satisfying a more complex relation that is very hard to check analytically (due to the large number of terms involved). It can be verified easily using **DiracQ** , providing a nontrivial demonstration. Another example illustrates the higher conservation laws of the 1-d Hubbard model that are non quadratic in the Fermi operators.

DiracQ can also be used in the initial stages of numerical calculations on finite systems, since it can generate the numerical Hamiltonian matrix on a given cluster. This is illustrated in a 4 site cluster with nearest and next nearest hopping bonds in the sector of 2 up and 2 down particles. This example also illustrates the strategies for generation of states using the Bra and Ket objects of **DiracQ** .

♦ 3. Contacts:

The authors can be contacted with comments and suggestions as well as further user generated examples of **DiracQ** by many channels. The preferred one is the email address for this purpose :

jgwright1986@gmail.com b.sriram.shastry@gmail.com

♦ 4. Suggested Acknowledgement of DiracQ

If **DiracQ** is useful in obtaining results leading to publication, a citation would be appropriate and appreciated.

Suggested Citation :
This work contains results obtained by using *DiracQ: A Quantum Many-Body Physics Package* , J. G. Wright and B. S. Shastry, arXiv:1301. (2013).

♦ 5. Acknowledgement:

The work at UCSC was supported by the US Department of Energy (DOE), Office of Science, Basic Energy Sciences (BES) under award number FG02-06ER46319.

DiracQ Example Book

A Compilation of Problems in Quantum Theory Performed Using the DiracQ Package

The following problems demonstrate how the DiracQ package can be used to solve non-trivial problems. Throughout this notebook ensure that you have the relevant operators activated on the DiracQ palette before evaluating any input.

```
SetDirectory[NotebookDirectory[]];
Get["DiracQ_V1.m"];
```

(I) Canonical Pairs

The package contains canonical position and momentum operators. Their use is demonstrated below for some simple examples.

```
? p
```

p is the canonical momentum operator. This operator can be called with one argument, taken
to be site index, or two arguments. The second argument will be taken to be coordinate direction.
Also included is the 3 dimensional canonical momentum vector, represented by OverVector[p], or $\vec{p}$.

```
? q
```

q is the canonical position operator. This operator can be called with one argument, taken
to be site index, or two arguments. The second argument will be taken to be coordinate direction.
Also included is the 3 dimensional canonical position vector, represented by OverVector[q], or $\vec{q}$.

The physical system can be in arbitrary dimensions and may refer to an arbotrary number of particles. The canonical pairs are endowed with indices: either one or two indices can be used. The first index is taken to be site label. The second index is taken to be the component. If only one index is given, a 1-dimensional system is assumed. This is demonstrated below. [Note: *unless you choose to declare q and p as active operators in the DiracQ Palette by ticking the boxes in the Palette, (or more aggressively turn all symbols on), the following commutators will turn out to vanish. If that happens, go back and turn them "on".*]

```
Commutator[p[j], q[i]]
```

```
-i ℏ δ[i, j]
```

```
Commutator[p[j, α], q[i, β]]
```

```
Commutator[p[j, y], q[i, y]]
```

```
-i ℏ δ[i, j]
```

```
Commutator[p, q]
```

```
0
```

Comment: In the last output, notice that p and q without any indices are treated as c-numbers. This can be a slight nuisance, but one resolution is to treat a set of harmonic oscillators indexed by a "i".

```
Clear[i, j]
```

$$\mathbf{H} = \frac{\mathbf{p[i]}^2}{2\,\mathbf{m}} + \frac{\mathbf{m}\omega^2}{2}\,\mathbf{q[i]}^2;$$

```
Commutator[H, p[i]]
```

$$i\ \left(m\omega^2\ \hbar\right)\ **\ q[i]$$

Next we construct creation and destruction operators from the canonical pairs.

```
a[i_] := Sqrt[m ω / (2 ħ)] (q[i] + i p[i] / (m ω));
a†[i_] := Sqrt[m ω / (2 ħ)] (q[i] - i p[i] / (m ω));
Commutator[a[i], a†[i]]
```

$$1$$

```
n[i_] := a†[i] ** a[i]
```

Here we show that [n[i], a[i]] = -a[i] and similarly [n[i],a†[i]]=a†[i]. SimplifyQ puts the expressions in a standard form where the identity of two expressions becomes clear.

```
SimplifyQ[a[i]]
```

$$\frac{\sqrt{\frac{m\omega}{\hbar}}\ **\ q[i]}{\sqrt{2}} + \frac{i\ \frac{1}{\sqrt{\frac{m\omega}{\hbar}}\ \hbar}\ **\ p[i]}{\sqrt{2}}$$

```
SimplifyQ[Commutator[n[i], a[i]] + a[i]]
```

$$0$$

(Note that the command is more explicitly A1= Commutator[n[i],a[i]]]; A2= +a[i]; A = SimplifyQ[A1+A2]. Of course the final result A vanishes, as it must)

```
SimplifyQ[Commutator[n[i], a†[i]] - a†[i] ]
```

$$0$$

(II) Hydrogen atom and Runge Lenz Laplace vectors

Here we define the Hamiltonian of the Hydrogen atom. We demonstrate the conservation of the Runge-Lenz-Laplace vector by using the package.
We use the three dimensional vector notation for the canonical pair. For example

```
q⃗[i]
q⃗[i].q⃗[i]
Clear[H];
H = p⃗[i].p⃗[i] / (2 m) - Z e^2 / Sqrt[q⃗[i].q⃗[i]];
? NCcross
L⃗[i_] = NCcross[q⃗[i], p⃗[i]]
Commutator[L⃗[i][[1]], H]
Commutator[L⃗[i][[2]], H]
SimplifyQ[Commutator[L⃗[i][[3]], H]]
```

The answer is so simple that it looks as though we have not performed any operation whatsoever. However, if we leave basic commutators unevaluated we see that indeed the computation was not

entirely trivial (feel free to try it by setting “Apply Definition” to “False” using the palette and then reevaluate the input). As an example consider the product q[i,y]**p[i,z], it is not conserved and we get a messy answer:

```
Commutator[q[i, y] ** p[i, z], H]
```

We next define the Runge Lenz vector $\vec{A}$[i] with a formal definition:

$$\vec{A}[i] = \vec{L}[i] \times \vec{p}[i] + Z e^2 m \frac{\vec{q}[i]}{\sqrt{\vec{q}[i].\vec{q}[i]}}$$

Using the NCcross function and symmetrizing in the two variables L and p, we work with

$$\vec{A}[i] = \frac{1}{2} \text{NCcross}\left[\vec{L}[i], \vec{p}[i]\right] - \frac{1}{2} \text{NCcross}\left[\vec{p}[i], \vec{L}[i]\right] + Z e^2 m \frac{\vec{q}[i]}{\sqrt{\vec{q}[i].\vec{q}[i]}};$$

We check a component of A to make sure it makes sense.

```
A[i][[3]]
```

```
term1 = Commutator[A[i][[3]], H]
```

With this expression, SimplifyQ does not produce a vanishing result since the expression on right locates p[i,z] differently in various terms and one needs to push this term to one side in ALL terms. Therefore at this stage it is better to push all the momentum type variables to the right extreme (the left extreme works equally well), so that all operators are ordered in a similar way. After that we can set all non commutative products to simple products and simplify. The function pp and ppp achieve this ordering, and sim does the job of simplifying the expressions after replacing the **’s with simple *’s.

```
pp[a_, m_] := PushOperatorRight[a, p[i, m]]
```

```
ppp[a_] := pp[pp[pp[a, x], y], z]
```

```
sim[a_] := Simplify[ppp[a] /. {NonCommutativeMultiply → Times}]
```

```
sim[term1]
```

This can be examined more closely by examining the intermediate terms:

```
ppp[term1] /. {NonCommutativeMultiply → Times}
```

```
Simplify[%]
```

The same scheme is demonstrated by pushing operators to the left (rather than right) here.

```
ppl[a_, m_] := PushOperatorLeft[a, p[i, m]]
```

```
pppl[a_] := ppl[ppl[ppl[a, x], y], z]
```

```
siml[a_] := Simplify[pppl[a] /. {NonCommutativeMultiply → Times}]
```

```
siml[term1]
```

(III) Fermions at work

- **For this Section Ensure that Fermionic Operators as well as Bra and Ket Vectors are Activated.**

Here we use 1 to represent spin up and - 1 to represent spin down. We now perform some test operations in the context of three problems using the Hubbard model Hamiltonian, given below.

```
Clear[i, j, n, H];
H = -∑_i ∑_j ∑_σ t[i, j] f†[i, σ] ** f[j, σ] + U ∑_i n_f[i, 1] ** n_f[i, -1];
```

Problem 1 : $[f_i^\sigma, H]$

These examples require the user to input expressions carefully. For example, notice that we have a sum over σ in the first term of H above. We therefore cannot use an operator that has a spin specified by σ, because the package does not differentiate that one sigma is inside the sum and the other is outside. It will attempt to sum over both σ's. This is not so much a restriction as it is a warning against sloppy notation that could easily be understood by a human but will be misinterpreted by the package. The same restriction applies to site indices, such as i and j in this example. The example below demonstrates incorrect usage of indices; we are taking the commutator of a fermi operator with a site index that is identical to the indice of summation in the Hamiltonian.

```
Commutator[f[i, σ], H]
```

$$\sum_i (U\,\delta[1, \sigma])\ **\ n_f[i, -1]\ **\ f[i, 1] -$$
$$\sum_i (U\,\delta[-1, \sigma])\ **\ f\dagger[i, 1]\ **\ f[i, -1]\ **\ f[i, 1] - \sum_i \sum_j \sum_\sigma t[i, j]\ **\ f[j, \sigma]$$

The above answer is **not** correct.

Trying same the problem again using proper notation where the external indices are different from the ones used in H, for example evaluate

```
Commutator[f[k, σ1], H]
```

$$(U\,\delta[1, \sigma 1])\ **\ n_f[k, -1]\ **\ f[k, 1] -$$
$$(U\,\delta[-1, \sigma 1])\ **\ f\dagger[k, 1]\ **\ f[k, -1]\ **\ f[k, 1] - \sum_j t[k, j]\ **\ f[j, \sigma 1]$$

we get the correct answer. We can simplify the answer further by specifying a value for $\sigma 1$.

```
Commutator[f[k, σ1], H] /. σ1 → 1
```

$$U\ **\ n_f[k, -1]\ **\ f[k, 1] - \sum_j t[k, j]\ **\ f[j, 1]$$

The best way of performing this calculation is just to start with the spin specified. In general this method is preferred because the package is then able to organize the operators using the value of the spins. It may therefore be necessary to reorder the output again if you specify the value of a spin after calculation.

```
Commutator[f[k, 1], H]
```

```
Commutator[f[k, -1], H]
```

Problem 2 : $[f_i^\sigma\, n_i^{-\sigma}, H]$

In this example we see that the answer is different in terms of operator ordering, depending on which method we use to perform the calculation.

```
Commutator[f[k, σ1] ** n_f[k, -σ1], H] /. σ1 → 1
```

$$U\ **\ f\dagger[k, -1]\ **\ f[k, -1]\ **\ f[k, 1] - U\ **\ f\dagger[k, -1]\ **\ n_f[k, 1]\ **\ f[k, -1]\ **\ f[k, 1] -$$
$$\sum_j t[k, j]\ **\ n_f[k, -1]\ **\ f[j, 1] + \sum_i t[i, k]\ **\ f\dagger[i, -1]\ **\ f[k, -1]\ **\ f[k, 1] -$$
$$\sum_j t[k, j]\ **\ f\dagger[k, -1]\ **\ f[j, -1]\ **\ f[k, 1]$$

```
Commutator[f[k, 1] ** n_f[k, -1], H]
```

$$U ** n_f[k, -1] ** f[k, 1] - \sum_j t[k, j] ** n_f[k, -1] ** f[j, 1] +$$
$$\sum_i t[i, k] ** f\dagger[i, -1] ** f[k, -1] ** f[k, 1] -$$
$$\sum_j t[k, j] ** f\dagger[k, -1] ** f[j, -1] ** f[k, 1]$$

The two results are not identical. However, if we reorder the operators in the top output we replicate the bottom output, which has the operators correctly ordered.

```
StandardOrderQ[Commutator[f[k, σ1] ** n_f[k, -σ1], H] /. σ1 → 1]
```

$$U ** n_f[k, -1] ** f[k, 1] - \sum_j t[k, j] ** n_f[k, -1] ** f[j, 1] +$$
$$\sum_i t[i, k] ** f\dagger[i, -1] ** f[k, -1] ** f[k, 1] -$$
$$\sum_j t[k, j] ** f\dagger[k, -1] ** f[j, -1] ** f[k, 1]$$

Performing the same calculation with a spin down particle we obtain the same answer with the spins reversed.

```
Commutator[f[k, -1] ** n_f[k, 1], H]
```

$$-U ** f\dagger[k, 1] ** f[k, -1] ** f[k, 1] - \sum_i t[i, k] ** f\dagger[i, 1] ** f[k, -1] ** f[k, 1] +$$
$$\sum_j t[k, j] ** f\dagger[k, 1] ** f[j, -1] ** f[k, 1] +$$
$$\sum_j t[k, j] ** f\dagger[k, 1] ** f[k, -1] ** f[j, 1]$$

Problem 3 : $H\, f^{\dagger\uparrow}_1 f^{\dagger\downarrow}_2 \left|\text{Vacuum}\right\rangle$

This demonstrates the properties of the Ket and Bra states, which need to be activated on the palette in addition to the Fermi, Bose or other relevant operators.
We can perform the product directly.

```
H ⊗ f†[1, 1] ** f†[2, -1] ** Ket[Vacuum]
```

$$\sum_i t[i, 1] ** f\dagger[2, -1] ** f\dagger[i, 1] ** \left|\text{Vacuum}\right\rangle +$$
$$\sum_i t[i, 2] ** f\dagger[i, -1] ** f\dagger[1, 1] ** \left|\text{Vacuum}\right\rangle$$

We see that the destruction operators annihilate the vacuum state

```
f[i, σ] ** Ket[Vacuum]
```

```
0
```

We could use the ProductQ operator here, or equivalently use the symbol ⊗ between the operator and the ket.

```
ProductQ[H, Ket[Vacuum]]
```

```
0
```

```
Ket[1] = f†[1, 1] ** f†[2, -1] ** Ket[Vacuum]
```

$$f\dagger[1, 1] ** f\dagger[2, -1] ** \left|\text{Vacuum}\right\rangle$$

```
ProductQ[H, f†[1, 1] ** f†[2, -1] ** Ket[Vacuum]]
```

$$\sum_i t[i, 1] ** f\dagger[2, -1] ** f\dagger[i, 1] ** |\text{Vacuum}\rangle + \sum_i t[i, 2] ** f\dagger[i, -1] ** f\dagger[1, 1] ** |\text{Vacuum}\rangle$$

We thereby see that all destruction operators have been killed and the correct answer is obtained. Clearly the same answer is found if we use Ket[1]

```
ProductQ[H, Ket[1]]
```

$$\sum_i t[i, 1] ** f\dagger[2, -1] ** f\dagger[i, 1] ** |\text{Vacuum}\rangle + \sum_i t[i, 2] ** f\dagger[i, -1] ** f\dagger[1, 1] ** |\text{Vacuum}\rangle$$

```
H ⊗ Ket[1]
```

$$\sum_i t[i, 1] ** f\dagger[2, -1] ** f\dagger[i, 1] ** |\text{Vacuum}\rangle + \sum_i t[i, 2] ** f\dagger[i, -1] ** f\dagger[1, 1] ** |\text{Vacuum}\rangle$$

Here we see that the symbol ⊗ could be used in place of explicitly typing out ProductQ[H,Ket[1]]. Trying the above problem w/o ⊗,

```
H ** f†[1, 1] ** f†[2, -1] ** Ket[Vacuum]
```

Note that we would not get the correct answer if we just used NonCommutativeMultiply instead of ⊗. Where it is appropriate to use NCM (**) rather than (⊗) might be confusing initially, but generally it is easy to deduce which one is appropriate. A rule of thumb is to useNCM in between single terms, and to use ⊗ in between larger expression involving many terms, Plus , Minus, or Sum functions).

(IV A) Hubbard Model k - space Moment Calculations

```
Clear[H]; H[l_, σ_, p_, q_, r_] := ∑_l ∑_σ ξ[l] f†[l, σ] ** f[l, σ] +
  (U / Ns) ∑_p ∑_q ∑_r f†[p + q, 1] ** f[p, 1] ** f†[r - q, -1] ** f[r, -1]
```

This is the Hubbard Hamiltonian in k space, with Ns as the number of sites, and the labels p,q,l, r as the wavevectors. and ξ=ϵ(k)- μ, with μ as the chemical potential and band dispersion ϵ(k). The dummy variables summed over in the right hand side (p,q,r,l,σ) are declared on the left hand side explicitly.

Throughout this notebook it is advisable to explicitly define the spin of the operators we are commuting with the Hamiltonian. This allows delta functions to be cancelled and thereby simplifies the result. Below we have computed the first moment using both spin up and spin down operators (1 for spin up, -1 for spin down). Also remember we must use new variables every time we commute with another Hamiltonian.

We now compute the symmetric moments of the Hubbard model- these are defined as $\omega_m = (-1)^m < \{[H, [H, \ldots [H, f], f\dagger\} >$ with m nested commutators. Thus explicitly the first moment is the expectation: $\omega_1(k) = (-) < \{[H, f(k, \sigma)], f\dagger(k, \sigma)\} >$. Below we work out the operators that emerge from this nested commutator and the final anticommutator. (Clearly the expectation of these requires the solution of the eigensystem that we do not address).
A good reference for these moments is W. Nolting, Z. Physik 255, 25-- 39 (1972), his Eqs(8-11) correspond to our moments below after shifting $\omega_m = M^{(m+1)}$.

(IV B) Hubbard Model r - space Moment Calculations

(V) Spins at work: Eight Vertex Model Yang - Baxter Equation

- **Activate the Pauli matrix symbol "σ"**

We use DiracQ to simplify some algebra that arises in the well known work of Baxter on the two dimensional 8 vertex model (R. J. Baxter, Ann. Phys. vol 76, p1 (1973)) . The transfer matrix of the model with a parameter u is written as:

$$T(u) = \underset{g}{\mathrm{Tr}}[L_{N,g}(u)\, L_{N-1,g}(u) \ldots L_{1,g}(u)]\,, \tag{1}$$

where each of the objects $L_{n,g}(u)$ is *a* scattering matrix

$$L_{n,g}(u) = \frac{a+b}{2} + \frac{a-b}{2}\sigma_n^z \sigma_g^z + \frac{c+d}{2}\sigma_n^x \sigma_g^x + \frac{c-d}{2}\sigma_n^y \sigma_g^y. \tag{2}$$

Here the Pauli matrices are defined for sites n=1,N and an extra “ghost” site “g”. The Boltzmann weights a, b, c, and d are functions of the spectral parameter u. The fundamental commutation relation of the transfer matrices [T(u),T(v)]=0 defines the integrability of this model and also is the key to its solution. Baxter shows that this commuting transfer matrix relation is satisfied if the local relation

$L_{n,g_2}[u] \otimes L_{n,g_1}[u'] \otimes R_{g_1,g_2}[u''] = R_{g_1,g_2}[u''] \otimes L_{n,g_1}[u'] \otimes L_{n,g_2}[u]$ (Baxter’s equation)

is satisfied, where explicitly

$$\begin{aligned} L_{n,g_2}(u) &= \frac{a+b}{2} + \frac{a-b}{2}\sigma_{g_2}^z \sigma_n^z + \frac{c+d}{2}\sigma_{g_2}^x \sigma_n^x + \frac{c-d}{2}\sigma_{g_2}^y \sigma_n^y \\ L_{n,g_1}(u') &= \frac{a'+b'}{2} + \frac{a'-b'}{2}\sigma_{g_1}^z \sigma_n^z + \frac{c'+d'}{2}\sigma_{g_1}^x \sigma_n^x + \frac{c'-d'}{2}\sigma_{g_1}^y \sigma_n^y \\ R_{g_2,g_1}(u'') &= \frac{a''+b''}{2} + \frac{a''-b''}{2}\sigma_{g_1}^z \sigma_{g_2}^z + \frac{c''+d''}{2}\sigma_{g_1}^x \sigma_{g_2}^x + \frac{c''-d''}{2}\sigma_{g_1}^y \sigma_{g_2}^y\,. \end{aligned} \tag{3}$$

Let us use the functions of the DiracQ package to see if we can find the requirements on the u's such that (10) is satisfied. For convenience let us set the three distinct sites in the problem explicitly as: `n=3, g2 = 1, g1 = 2.` This allows the code to automatically solve the Kronecker δ functions, as shown below. For more information on use of the Kronecker δ see the Tutorial.

```
δ[g1, g2]
```

```
δ[2, 1]
```

```
n = 3; g2 = 1; g1 = 2;
Ln,g2[W] =
  (a + b)/2 + (a - b)/2 σ[g2, z] ** σ[n, z] + (c + d)/2 σ[g2, x] ** σ[n, x] + (c - d)/2 σ[g2, y] ** σ[n, y];
Ln,g1[W'] = (a' + b')/2 + (a' - b')/2 σ[g1, z] ** σ[n, z] +
  (c' + d')/2 σ[g1, x] ** σ[n, x] + (c' - d')/2 σ[g1, y] ** σ[n, y];
Rg1,g2[W''] = (a'' + b'')/2 + (a'' - b'')/2 σ[g1, z] ** σ[g2, z] +
  (c'' + d'')/2 σ[g1, x] ** σ[g2, x] + (c'' - d'')/2 σ[g1, y] ** σ[g2, y];
```

Let us first compute the product of the LHS Baxter's equation. For this we will use the function, which will apply the algebra of Pauli matrices and collect c terms. Because there are 8 terms in each scattering matrix the number of terms on both the right and left hand sides of equation is large (128 terms in all). Because of this we will suppress the output.

```
LHS = Ln,g2[W] ⊗ (Ln,g1[W'] ⊗ Rg1,g2[W'']);

RHS = Rg1,g2[W''] ⊗ (Ln,g1[W'] ⊗ Ln,g2[W]);

{Length[LHS], Length[RHS]}
```

We can take a peek at the output:

```
Short[LHS, 3]

Short[RHS, 3]
```

We now need to collect terms which involve the same set of operators. We can then set the sum of the coefficients of all the terms with identical operators to zero. We will now use the function TakeQ-Part, to extract the string of operators from each term. By using the Union function we can also remove duplicate expressions. We are thereby left with a list of all the different strings of operators that appear anywhere throughout the expression.

```
OperatorTerms = Union[TakeQPart[LHS - RHS]]

Length[OperatorTerms]
```

We can now use the QCoefficient function to find the coefficients of a string of operators. The use of this function is demonstrated below.

```
QCoefficient[LHS - RHS, OperatorTerms[[1]]]
```

The function has found all of the terms with operators that match the pattern of OperatorTerms[[1]] and summed their coefficients. This term must go to zero for the Yang - Baxter relation to hold. We can work this function into a loop and find all such expressions at once, as shown below.

```
Do[expression[n] = QCoefficient[LHS - RHS, OperatorTerms[[n]]];
 Print[expression[n] == 0], {n, Length[OperatorTerms]}]
```

These are the final equations. By recombining them and simplifying we reproduce Baxter's equations.

```
Expand[(expression[2] + expression[4]) / i] == 0
Expand[(expression[1] + expression[3]) / i] == 0
Expand[(expression[5] + expression[6]) / i] == 0
Expand[(expression[2] - expression[4]) / i] == 0
Expand[(expression[1] - expression[3]) / i] == 0
Expand[(expression[5] - expression[6]) / i] == 0
```

These are six homogeneous equations relating the twelve parameters (a,b,c,d) etc. Baxter proceeds to give an elegant analysis of these by parameterization in terms of the spectral parameters u, u', u''. The solution is well described in the original paper; we see above that DiracQ helps in automating the tedious calculations.

(VI) 1D Hubbard Model SSS Matrix Problem

- **Activate the Pauli matrix symbol "σ"**

$$H = \sum_n H_{n+1,n}; \quad H_{n+1,n} = (\sigma_n^+ \sigma_{n+1}^- + \sigma_{n+1}^+ \sigma_n^-) + (\tau_n^+ \tau_{n+1}^- + \tau_{n+1}^+ \tau_n^-) + \frac{1}{4} U \sigma_n^z \tau_n^z . \tag{4}$$

This corresponds to the quantum Hamiltonian, commuting with the transfer matrix of two Free fermi models coupled in a specific way. We start with a single Pauli species scattering matrix :

$$l_{ij}^{(\sigma)}(\theta) = \frac{a+b}{2} + \frac{a-b}{2} \sigma_i^z \sigma_j^z + \frac{c}{2} (\sigma_i^x \sigma_j^x + \sigma_i^y \sigma_j^y) \tag{5}$$

Here the parameters a b c satisfy the Free Fermi condition $a^2 + b^2 = c^2$. This is conveniently parametrized by setting
$a = c \operatorname{Cos}(\theta)$ $b = c \operatorname{Sin}(\theta)$. For the two sites 1 and 2 we define
$a_1 = \operatorname{Cos}[\theta_1]$, $b_1 = \operatorname{Sin}[\theta_1]$, $a_2 = \operatorname{Cos}[\theta_2]$, $b_2 = \operatorname{Sin}[\theta_2]$ and denote $(W_j \equiv (\theta_j, h_j))$.

We may write two sets of scattering matrices $l_{ij}^{(\sigma)}$ and $l_{ij}^{(\tau)}$ for the two species at the same θ :

$$l_{ij}(\theta) = l_{ij}^{(\sigma)}(\theta) \otimes l_{ij}^{(\tau)}(\theta) \tag{6}$$

The scattering operator for the Hubbard model is constucted as

$$L_{3\,n}(W_n) = l_{3\,n}(\theta_n)\, I_n(h_n),$$

The transfer matrix can now be written as

$$T(W) = \operatorname{Tr}_o[L_{No}(W)\, L_{N-1\,o}(W) \ldots L_{1\,o}(W)]. \tag{7}$$

With these definitions, the transfer matrix forms a commuting family $[T(W_1), T(W_2)] = 0$ provided the scattering matrix is proven to satisfy the Yang – Baxter equation $(W_n \equiv (\theta_n, h_n))$

$$L_{31}(W_1)\, L_{32}(W_2)\, S_{12}(W_2 \mid W_1) = S_{12}(W_2 \mid W_1)\, L_{32}(W_2)\, L_{31}(W_1). \tag{8}$$

The S operator found by Shastry is given by:

$$S_{12}(W_2 \mid W_1) = \rho[\cos(\theta_2 + \theta_1) \cos(h_2 - h_1)\, l_{12}(\theta_2 - \theta_1) + \cos(\theta_2 - \theta_1) \sinh(h_2 - h_1)\, l_{12}(\theta_2 + \theta_1)\, \sigma_2^z \tau_2^z]. \tag{9}$$

An extra and crucial sufficiency condition is that the fields h_n $(n = 1, 2)$ must satisfy

$$\frac{\operatorname{Sinh}[2 h_n]}{a_n b_n} = \frac{U}{2}, \tag{10}$$

where U is the interaction term in the Hubbard Hamiltonian. The peculiarity of this model is that $S(W_2 \mid W_1)$ depends on the spectral parameters W_1 and W_2 separately. It cannot be written in terms of the difference of the spectral parameters, unlike most other standard models. This relation was proved by Shastry analytically in B. S. Shastry, J. Stat. Phys. 50, 57 (1988).

$$T = \operatorname{Tr}_o[S_{No}(W_1 \mid W_2)\, S_{N-1\,o}(W \mid W_{N-1}) \ldots S_{1\,o}(W \mid W_1)]. \tag{11}$$

so that on setting W_j to 0, we get back the Hubbard transfer matrix above. The inhomogeneous matrix T commutes for different weights according to

$$[T(W \mid W_1 \ldots W_N), T(W' \mid (W_1 \ldots W_N)] = 0. \tag{12}$$

The sufficiency condition for this is the more general Yang Baxter relation

$$S_{31}(W_1 \mid W_3)\, S_{32}(W_2 \mid W_3)\, S_{12}(W_2 \mid W_1) = S_{12}(W_2 \mid W_1)\, S_{32}(W_2 \mid W_3)\, S_{31}(W_1 \mid W_3). \tag{13}$$

This is Eq (4.14) of the paper and several typical matrix elements were verified by Shastry by explicit calculation, who conjectured that this equation is true for all matrix elements. Since the number of terms is very large (see below), an exhaustive enumerative proof

was not given. We will provide such an enumerative proof using DiracQ next.

For non zero W_3, analytical results in this direction were found much later in the work of M Shiroishi and M Wadati, Journal of Physical Society of Japan, Vol 64, 2795 (1995). The calculation below demonstrates the power of DiracQ quite well. The large number of terms involved make the algebra prohibitive by a hand calculation for most of us! Using the package however we are able to compute the product of three S operators in a number of minutes.

The first step is to input the form for the scattering matrix and for the S operators. Here, we denote the two separate species of Pauli matrices not by use of another symbol, but rather by using a third index to denote the species of the operator, as shown below.

$$\sigma_1^z \to \sigma[1, z, 1], \ \tau_1^z \to \sigma[1, z, 2] \tag{14}$$

In this notation the scattering matrix and the S operator are written below as general functions.

```
Clear[A, l, S];
l_{u_,v_}[n_, p_, q_] :=
  (A[n, 1, p, q] + A[n, -1, p, q] σ[v, z, 1] ** σ[u, z, 1] + σ[v, x, 1] ** σ[u, x, 1] +
     σ[v, y, 1] ** σ[u, y, 1]) ⊗ (A[n, 1, p, q] + A[n, -1, p, q] σ[v, z, 2] ** σ[u, z, 2] +
     σ[v, x, 2] ** σ[u, x, 2] + σ[v, y, 2] ** σ[u, y, 2]);
S_{u_,v_} := Cos[θ_v + θ_u] Cosh[h_v - h_u] l_{u,v}[-1, θ_v, θ_u] +
  Cos[θ_v - θ_u] Sinh[h_v - h_u] l_{u,v}[1, θ_v, θ_u] σ[v, z, 1] σ[v, z, 2]
```

The Boltzmann weights in the scattering matrix are not written explicitly because this increases the number of terms involved in the product and their definition will not be relevant until after the S operator products have been calculated. They are therefore represented by the function A, which will be defined after the product of the three S operators on each side of (36) has been computed. We will set

```
rulesimplify = { h_i_ -> 1 / 2 ArcSinh[U / 2 Cos[θ_i] Sin[θ_i]], ,
    A[n_, m_, p_, q_] -> Cos[p + n q] + m Sin[p + n q]}
```

after the calculation with the operators is performed, and it is easy to see that this replacement reproduces the scattering operator above with a scale factor of 2 which cancels out. Here we compute the product of three S , matrices. Each S matrix contains about thirty terms, we we wexpect the product to contain on the order of 27,000 terms. This expression is so large as to be prohibitive to compute by hand.

```
{time, LHS} = Timing[S_{3,1} ⊗ (S_{3,2} ⊗ S_{1,2})];
```

```
time / 60
```

The timing function informs us that this operation takes around three minutes on our testing machine.

```
Length[LHS]
```

We see that there are approximately 32,000 terms in the final product. Below an arbitrary example of a single term is shown. We expect similar timing and length for the RHS.

```
LHS[[6]]
```

We now compute the RHS

```
RHS = S_{1,2} ⊗ (S_{3,2} ⊗ S_{3,1});
```

```
Length[RHS]
```

We have now computed both the left and right hand sides of equation 13. If the equation holds, they are equal. We verify this by subtracting the two and returnign zero.

```
subtraction = LHS - RHS;
```

```
Length[subtraction]
```

We would have hoped that when we subtracted the LHS from the RHS that the result was zero. It appears we will have to do more manipulation to show that the LHS is equal to the RHS. We now

have a very large expression, and we need to manipulate it in several ways. We need to find all distinct strings of operators and then find the coefficient of all of each string. We must then show that every one of these coefficients vanishes. DiracQ contains functions that will perform the first two. However, were we to use the functions outright the computation time would be very long. Every expression input into a DiracQ function is first organized into lists that can be manipulated (see the ' Brief Explanation of Form' section). For a large expression such as the one we are working with, organizing the expression takes a very long time. The expression must then be recompiled into and understandable form. We can circumvent some of these steps by organizing an expression once and specifiying to subsequent functions that the input has already been organized and that this step should be skipped. This is done by specifying' Organized Expression -> True' as an option. This process is demonstrated below.

```
? Organize
```

```
{time, organizedsubtraction} = Timing[Organize[subtraction]];
```

```
time / 60
```

We see that on our test machine the Organize function takes a few minutes to evaluate and is computationally quite expensive. Below we use the TakeQPart function to find the strings of operators that appear in OrganizedSubtraction, specifying that the input is already an organized expression to save some computational time.

```
OperatorTerms =
  Union[TakeQPart[organizedsubtraction, OrganizedExpression → True]];
```

```
Length[OperatorTerms]
```

```
OperatorTerms[[300]]
```

The above output shows that there are 472 distinct strings of operators found in the expression and an example of one such string. We can now use the QCoefficient function to find the coefficients of a string of operators.

```
? QCoefficient
```

The result is not printed due to the length and complexity, though we do show the first term in the expression. See the appendix section, or remove the semicolon at the end to see the expression.

```
sum[1] = QCoefficient[organizedsubtraction,
   OperatorTerms[[1]], OrganizedExpression → True];
```

```
Length[sum[1]]
```

```
sum[1][[1]]
```

We now apply the definition of the Boltzmann weights.

```
rulesimplify = { h_i_ -> 1 / 2 ArcSinh[U / 2 Cos[θ_i] Sin[θ_i]],
   A[n_, m_, p_, q_] -> Cos[p + n q] + m Sin[p + n q]};
```

```
Simplify[sum[1] /. rulesimplify]
```

We see that the coefficients of this one string of operators simplify to zero. We must show that each such term vanishes for all 472 strings of operators. The command below will evaluate each such term.

```
Do[sum[i] = Simplify[QCoefficient[organizedsubtraction, OperatorTerms[[i]],
     OrganizedExpression → True] /. rulesimplify], {i, 1, 472}]
```

```
Table[sum[i], {i, 1, 472}]
```

Thus we see that every term simplifies to zero and thus proves the result.

(VII) Hubbard Model Currents

Below we define the Hamiltonian of the Hubbard model using fermionic operators. The system is taken to have periodic boundary conditions such that (N + a) a where N is the number of sites. We also write two currents which have been shown to commute with the Hamiltonian in the papers B.S.Shastry Phys. Rev. Letts. vol 56, 1529, 2453 (1986) and especially J.Stat.Phys, vol 50, 57 (1988). Here it is shown that for a system with odd number of sites we have a conservation law termed j_A (defined below) that stands apart from the other conservation laws. A current j_B is also found (defined below) and unlike j_A generalizes to many more currents since it is the first non trivial member of the currents contained in the transfer matrix. Note that j_A corresponds to infinite ranged hopping of the doubly occupied sites adding to short ranged hops of fermions, whereas j_B corresponds to second neighbor hopping.

We will now verify the commutation of j_A for odd number of sites using DiracQ and the commutation of j_B for both odd and even sites. Notice that we write the Hamiltonian as well as the currents as functions of the number of sites NN (N is a predefined symbol in *Mathematica*). This will allow us to vary the number of sites between even and odd values.

```
Clear[H]; Remove[j];
H[NN_] := -t ∑_{n=1}^{NN} Sum[(f†[n + 1, σ] ** f[n, σ] + f†[n, σ] ** f[n + 1, σ]), {σ, -1, 1, 2}] +
   U ∑_{m=1}^{NN} n_f[m, 1] ** n_f[m, -1]
j_A[NN_] := i t ∑_{l=1}^{NN} Sum[(-f†[l, σ] ** f[1 + l, σ] + f†[1 + l, σ] ** f[l, σ]), {σ, -1, 1, 2}] +
    i U ∑_{r=1}^{NN} ∑_{l=1}^{NN-1} (-1)^l f†[l + r, 1] ** f†[l + r, -1] ** f[r, -1] ** f[r, 1];
j_B[NN_] := i t ∑_{o=1}^{NN} Sum[f†[o + 2, σ] ** f[o, σ] - f†[o, σ] ** f[o + 2, σ], {σ, -1, 1, 2}] +
    i U ∑_{o=1}^{NN} Sum[f†[o + 1, σ] ** f[o, σ] - f†[o, σ] ** f[o + 1, σ], {σ, -1, 1, 2}] +
    i U ∑_{o=1}^{NN} Sum[(f†[o, σ] ** (f[o + 1, σ] - f[o - 1, σ]) -
          (f†[o + 1, σ] - f†[o - 1, σ]) ** f[o, σ]) n_f[o, -σ], {σ, -1, 1, 2}];
```

First we will perform the commutator of the Hamiltonian with j_A for a system with 5 sites. As the number of sites increases so does the number of operations performed, and therefore we will only perform these commutators for relatively small systems. Even so the computing time is nonnegligible. We must include the periodic boundary conditions of our system. These are written as replacement rules. The replacement rules must be specified to each term, and cannot be specified at the end of computation, as this will lead to an erroneous answer.

```
PBCrule[n_] := {f[i_, a_] /; i > n → f[i - n, a], f[i_, a_] /; i < 1 → f[i + n, a],
   f†[i_, a_] /; i > n → f†[i - n, a], f†[i_, a_] /; i < 1 → f†[i + n, a]}
Timing[Commutator[H[5] /. PBCrule[5], j_A[5] /. PBCrule[5]]]
```

Performing the same computation with a four site sytem we see that the result is nonvanishing, as we expected.

```
Commutator[H[4] /. PBCrule[4], j_A[4] /. PBCrule[4]]
```

Again, a system with an odd number of sites commutes with the Hamiltonian.

```
Timing[Commutator[H[7] /. PBCrule[7], j_A[7] /. PBCrule[7]]]
```

We now verify that j_B commutes with the Hamiltonian for both odd and even number of sites.

```
Timing[Commutator[H[5] /. PBCrule[5], j_B[5] /. PBCrule[5]]]
```

```
Timing[Commutator[H[6] /. PBCrule[6], jB[6] /. PBCrule[6]]]
```

Here we investigate whether the two currents commute with each other. The results suggest that for odd N the two currents commute but that they do not commute for even N.

```
Timing[Commutator[jA[5] /. PBCrule[5], jB[5] /. PBCrule[5]]]
```

From the computation below we see that the commutator is non zero for even sites.

```
evensitecommutator = Commutator[jA[4] /. PBCrule[4], jB[4] /. PBCrule[4]];
```

```
Short[evensitecommutator, 2]
```

In view of the above results, it seems very likely that $[j_A, T] = 0$ for odd number of sites, where T is the transfer matrix of the Hubbard model in the Fermi representation, whereas the commutator is non zero for even sites.

(VIII) Construction of Cluster Hamiltonians of the Hubbard model using Bras and Kets

We illustrate the use of the Bra[Vacuum] and Ket[Vacuum] symbols, for creating the space of allowed states in the Hubbard model on a small cluster, and furhter to operate the Hamiltonian on these to generate its matrix representation. Let us consider a 4 site cluster defined in the diagram, with the Hubbard Hamiltonian hopping parameters (t1, t2 , U). For illustration we confine ourselves to the sector with Sz=0 and 4 particles, i.e. the half filled state, where the number of basis states is 36.

(1) t1 (2)

t1 t2 t1

(3) t1 (4)

4-Site Hubbard Cluster with nearest neighbour hops t1, and second neighbour hops t2.

```
hop[i_, j_] := - Sum[f†[i, σ] ** f[j, σ] + f†[j, σ] ** f[i, σ] , {σ, -1, 1, 2}]
```

```
pot[i_] := U nf[i, 1] ** nf[i, -1]
```

```
Hcluster = t1 ( hop[1, 2] + hop[2, 3] + hop[3, 4] + hop[4, 1]) +
    t2 ( hop[1, 4] + hop[2, 3]) + pot[1] + pot[2] + pot[3] + pot[4] ;
```

We now define the creation and destruction operators for pairs of sites with a given spin projection,

```
twofermi[i_, j_, σ_] = f†[i, σ] ** f†[j, σ];
twofermidest[i_, j_, σ_] = f[j, σ] ** f[i, σ];
```

Let us see the explicit form of a basis ket. The function StandardOrderQ organizes a Fermi operator product, its convention is to write these with increasing site labels from left to right, and puts the down spin blocks to the left of the up spin blocks.

```
StandardOrderQ[twofermi[1, 2, 1] ** twofermi[2, 3, -1] ** Ket[Vacuum]]
```

We next produce the 36 basis kets {Ψ[1], .., Ψ[36]} and their adjoint 36 basis bras {ΨB[1], .., ΨB[36]}

```
ii = 1;
Do[Ψ[ii] = StandardOrderQ[twofermi[i, j, 1] ** twofermi[k, l, -1] ** Ket[Vacuum]];
 ii = ii + 1, {i, 1, 4}, {j, 1, i - 1}, {k, 1, 4}, {l, 1, k - 1}];
ii = 1;
Do[ΨB[ii] =
   StandardOrderQ[Bra[Vacuum] ** twofermidest[i, j, 1] ** twofermidest[k, l, -1]];
 ii = ii + 1, {i, 1, 4}, {j, 1, i - 1}, {k, 1, 4}, {l, 1, k - 1}];
```

It is useful to inspect a typical basis state pair

```
Ψ[9]
```

```
ΨB[18]
```

We next illustrate the action of the Hubbard Hamiltonian on these states, here the function Standard-OrderQ does the job of simplifying expressions bu pushing the destruction operators to the extreme right and hitting the Ket[Vacuum], which is annihilated, or a similar story for the Bra[Vacuum] that is annihilated by creation operators. A brief glance at the output shows that we get back the basis states multiplied by coefficients that depend on t1,t2 and U.

```
Short[StandardOrderQ[Hcluster ** Ψ[9]], 3]
```

```
Short[StandardOrderQ[ΨB[18] ** Hcluster], 2]
```

At this point, there are two ways to construct the Hamiltonian matrix. Starting with expressions of the type StandardOrderQ[Hcluster**Ψ[i]], we can take the inner product with the bras ΨB[j]. As an alternative, we can pick off the coefficients using the QCoefficient function, this is faster and hence prefereable in most situatiions. We illustrate both methods and display the time saved by the second method. First we need a rule that collapses the fundamental innerproduct to unity:

```
ruleinnerproduct = {Bra[Vacuum] ** Ket[Vacuum] → 1};
```

```
Timing[Do[res[i] = StandardOrderQ[Hcluster ** Ψ[i]], {i, 1, 36}]]
```

The array res[i] stores the resultant of the action of the Hamiltonian on the ith ket.
The two methods to creat the Hamiltonian matrix use the innerproduct with the ΨB[j], i.e. the Bra vector, or picking out the coefficients of Ψ[j] in res[i]. These are called matrix1 and matrix2 respectively.

```
matrix1[j_, i_] := StandardOrderQ[ΨB[j] ** res[i]] /. ruleinnerproduct
matrix2[j_, i_] := QCoefficient[res[i], Ψ[j]]
```

```
R1 = Timing[Table[matrix1[k, i], {k, 1, 36}, {i, 1, 36}]];
R2 = Timing[Table[matrix2[k, i], {k, 1, 36}, {i, 1, 36}]];
```

```
R1[[1]]
```

```
R2[[1]]
```

```
Short[R1[[2]] - R2[[2]], 3]
```

We see that the two methods give the same resulting matrix, but the second method is considerably faster. We next define the ClusterHamiltonian and check its eigenvalues in some simple cases. For larger systems, it is clearly more advantageous to use the sparse matrix notation in such problems, but we will not pursue it here.

```
ClusterHamiltonian[t1_, t2_, U_] = R1[[2]];
```

```
Eigenvalues[ClusterHamiltonian[1, 0, 0]]
```

```
Eigenvalues[ClusterHamiltonian[0, 0, U]]
```

The first set of eigenvalues correspond to a non interacting model with only nearest neighbour hoppings, and is easily seen to be correct. The second set of eigenvalues correspond to the local

limit, where we have 6 states in the U=0 sector (Lowest Hubbard Band), 6 states in the U=2 sector (Second Hubbard band) and the rest with U=1, i.e. the First Hubbard Band.

Appendices

(I) Explanation of Form

The strength of the system lies in the ability to organize, rearrange, and manipulate terms in an expression based on their properties. The method of organization provides a simple language for the manipulation of terms. An understanding of the language used by the code is not essential for most operations. The user is only required to input algebraic expressions in a logical and minimally restricted form to effectively use the functions of this package. The language of organization will briefly be covered here for posterity.

Mathematica organizes all input in terms of lists. Each list has a head. The head of a list is the function that is to be applied to the list. The example below shows how *Mathematica* organizes some algebraic input into lists and functions.

```
FullForm[5 ∑i c[i] b[i] ** b†[j] + 2 ∑i d[i] b†[i] ** b[j]]
```

It is the job of the Organize function to scan through this list and extract the relevant information. The Organize function is therefore at the heart of every operation of the code. This function reads input such as the example above and organizes it based on the properties of the terms encountered. Information encountered by the Organize function is stored in nested lists. Different types of objects are placed in different places within the larger list created by the Organize function. Organize recognizes four types of objects : numbers, sums, constants, and operators. Each of these types of objects is placed in a separate nested list. The easiest way to understand this ordering is through example

```
Organize[5 ∑i c[i] b[i] ** b†[j]]
```

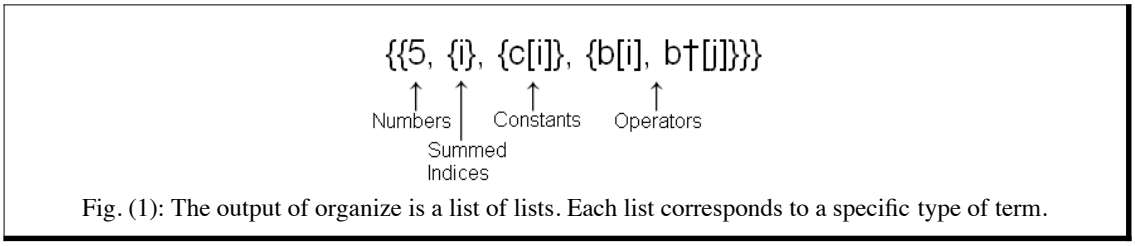

Fig. (1): The output of organize is a list of lists. Each list corresponds to a specific type of term.

```
Organize[5 ∑i c[i] b[i] ** b†[j] + 2 ∑i d[i] b†[i] ** b[j]]
```

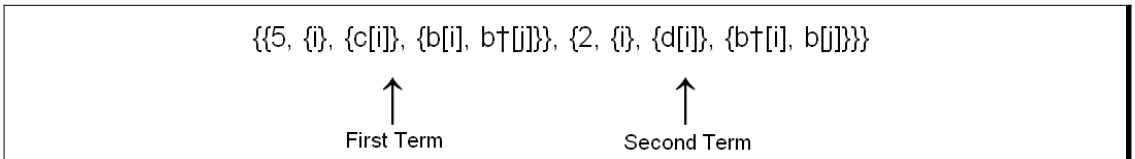

Fig.(2) : When the input of Organize is the addition of several terms,

each term is contained in a list within the larger list.

When a function is encountered that cannot be decomposed further, the Organize function determines whether this function contains any operators. If so, the package uses a special notation to signify that it has found a function of operators that cannot be decomposed further. The notation function[a, {b}] is used, where a is the function that cannot be decomposed, and b is a list of the operators on which the function depends. As it currently stands not all functions can be read and placed in this notation, but any function involving operators to different powers or exponential functions of operators can be understood. Using this format the user is free to define commutators of more complicated functions of operators as required. The notation used is identical to that used to add an operator, with the caveat that the definition will be written using the "function" notation described above. An example of an organized function such as described above is given below.

```
Organize[a ℮^(t[i] q[i]+q[j])]
```

Humanize is the functional opposite of organize in that Humanize reads the language created by the Organize function and recreates a mathematical form that a user can understand. This function is demonstrated below.

```
Humanize[{{5, {i}, {c[i]}, {b[i], b†[j]}}, {2, {i}, {d[i]}, {b†[i], b[j]}}}]
```

With this simple language the manipulation of certain types of terms can be performed with greater ease. Rather than manipulating operators as we come across them, we collect every operator in the expression. We can then perform algorithms on the lists of operators to combine and manipulate them as necessary. It is necessary to collect numbers and constants separately so that numbers can be combined as necessary and constants, which may depend on indices of summation, can be placed inside a sum. Also, the ability to identify summed indices allows us to evaluate delta functions. Every function in the package uses this form to organize input.

(II) Demonstration Problem Output (Produces large outputs hence not meant for printing)