Projeect 2

Charles Loelius

April 7, 2014

1 Hamiltonian for Pair Conservation

In this problem we consider a space of equally spaced single particle levels with a spin degeneracy(up and down spin). We can then consider a pair conserving Hamiltonian with a single particle part and a pairing term. We represent these as

$$H = H_0 + V \tag{1}$$

Where

$$H_0 = \chi \sum_{p\sigma} (p-1)\, a^\dagger_{p\sigma} a_{p\sigma} \tag{2}$$

$$\langle q_+ q_- | V | s_+ s_- \rangle = -g \rightarrow V = -g \sum_{pq} a^\dagger_{+p} a^\dagger_{p-} a_{q+} a_{q-} \tag{3}$$

Finally we define a pair creation and annhilation operator as

$$\hat{P}_p = a_{p+} a_{p-} \tag{4}$$

$$\hat{P}_p{}^\dagger = a^\dagger_{p+} a^\dagger_{p-} \tag{5}$$

From this it follows that

Now let us consider each of the commutation relations between the single particle and creation annhilation operators.

$$[P_p, P_q] = 0 \tag{6}$$

$$\left[P_p, P_q^\dagger\right] = \delta_{p,q} \tag{7}$$

$$[P_p, a_{q\sigma}] = 0 \tag{8}$$

$$\left[P_p^\dagger, a_{q\sigma}\right] = -\delta_{p,q} a_{p-\sigma}^\dagger \tag{9}$$

$$\left[P_p, a_{q\sigma}^\dagger\right] = \delta_{p,q} a_{p-\sigma} \tag{10}$$

$$\left[P_p^\dagger, a_{q\sigma}^\dagger\right] = 0 \tag{11}$$

$$\left[P_q^\dagger, P_p^\dagger\right] = 0 \tag{12}$$

We note that we can write V as

$$V = -g \sum_{pq} P_q^\dagger P_p \tag{13}$$

1.1 H_0, V commute withS_z, S^2

1.1.1 H_0 commutes with S_z

$$[H_0, S_z] = \epsilon \sum_{p'\sigma' p\sigma} (p-1)\,\sigma' \left[a_{p\sigma}^\dagger a_{p\sigma}, a_{p'\sigma'}^\dagger a_{p'\sigma'}\right] = A\left(\delta_{p,p'}\delta_{\sigma,\sigma'} - \delta_{p,p'}\delta_{\sigma,\sigma'}\right) = 0 \tag{14}$$

1.1.2 V commutes with S_z

$$[V, S_z] = A \sum_{p,q,p',\sigma} \left[P_p^\dagger P_q, a_{p'\sigma}^\dagger a_{p\sigma}\right] = \tag{15}$$

$$A \sum_{p,q,p',\sigma} \left(P_p^\dagger \left[P_p, a_{p'\sigma}^\dagger\right] a_{p'\sigma} + P_p^\dagger a_{p'\sigma}^\dagger \left[P_p, a_{p'\sigma}\right] + \left[P_p^\dagger a_{p'\sigma}^\dagger\right] P_p, a_{p'\sigma} + a_{p'\sigma}^\dagger \left[P_p^\dagger, a_{p'\sigma}\right] P_p\right) \tag{16}$$

$$= A \sum_{p,q,p'\sigma} \left(\delta_{p,p'} P_p^\dagger a_{p'-\sigma} a_{p'\sigma} - \delta_{p,p'} a_{p',\sigma}^\dagger a_{p,-\sigma}^\dagger P_p\right) = A \sum_{p\sigma} \left(P_p^\dagger P_p - P_p^\dagger P_p\right) = 0 \tag{17}$$

1.1.3 H_0 commutes with S^2

We note that

$$S^2 = S_z^2 + \frac{1}{2}\left(S_+ S_- + S_- S_+\right) \tag{18}$$

Knowing from above that H_0 commutes with S_z it thereby follows that we must only prove

$$[H_0, (S_+ S_- + S_- S_+)] = 0 \tag{19}$$

We note for any operator O

$$[O,(S_+S_- + S_-S_+)] = [O,S_+]\,S_- + [O,S_-]\,S_+ + S_+\,[O,S_-] + S_-\,[O,S_+] \tag{20}$$

We then consider that H_0 is some constant times $a^\dagger_{p\sigma}a_{p\sigma}$. Thus we consider that

$$\left[H_0, a^\dagger_{p'\sigma'}a_{p'-\sigma'}\right] = \sum_{p\sigma}(p-1)\left[a^\dagger_{p\sigma}a_{p\sigma}, a^\dagger_{p'\sigma'}a_{p'-\sigma'}\right] = \tag{21}$$

$$\sum_{p\sigma} p(p-1)\delta_{pp'}\left(\delta_{\sigma\sigma'}a^\dagger_{p'\sigma'}a_{p\sigma} - \delta_{\sigma-\sigma'}\right) = 0 \tag{22}$$

From this it immediately follows that

$$[H_0, S_\pm S_\mp] = 0 \rightarrow \left[H_0, S^2\right] = 0 \tag{23}$$

1.1.4 $[V, S_+S_- + S_-S_+ = 0]$

Again we take advantage of the general operator format, leading to

$$[V,(S_+S_- + S_-S_+)] = -g\textstyle\sum_{pq}\left(\left[P^\dagger_q P_p, S_+\right]S_- + \left[P^\dagger_q P_p, S_-\right]S_+ + S_+\left[P^\dagger_q P_p, S_-\right] + S_-\left[P^\dagger_q P_p, S_+\right]\right) = \tag{24}$$

$$-g\textstyle\sum_{pq}\left(\left(P^\dagger_q\left[P_p,S_+\right] + \left[P^\dagger_q,S_+\right]P_p\right)S_- + \left(P^\dagger_q\left[P_p,S_-\right] + \left[P^\dagger_q,S_-\right]P_p\right)S_+ + S_+\left(P^\dagger_q\left[P_p,S_-\right] + \left[P^\dagger_q,S_-\right]P^\dagger_q\right) + S_-\left(P^\dagger_q\left[P_p,S_+\right] + \left[P^\dagger_q,S_+\right]P_p\right)\right) \tag{25}$$

Next we consider

$$\left[P^\dagger_p, S_\sigma\right] = \left[P^\dagger_p, \sum_p a^\dagger_{p\sigma}a_{p-\sigma}\right] = \sum_p{}' \left(\left[P^\dagger_p, a^\dagger_{p'\sigma}\right]a_{p'-\sigma} + a^\dagger_{p'\sigma}\left[P^\dagger_p, a_{p'-\sigma}\right]\right) = -a^\dagger_{p-\sigma}a^\dagger_{p\sigma} \tag{26}$$

$$[P_p, S_\sigma] = \left[P_p, \sum_p a^\dagger_{p\sigma}a_{p-\sigma}\right] = \sum_p{}' \left(\left[P_p, a^\dagger_{p'\sigma}\right]a_{p'-\sigma} + a^\dagger_{p'\sigma}\left[P_p, a_{p'-\sigma}\right]\right) = a_{p-\sigma}a_{p\sigma} \tag{27}$$

Puting these together we are left with

$$-g\textstyle\sum_{pq}\left(\left(P^\dagger_q a_{p-}a_{p+} - a^\dagger_{q-}a^\dagger_{q+}P_p\right)S_- + \left(P^\dagger_q a_{p-}a_{p+} - a^\dagger_{q-}a^\dagger_{q+}P_p\right)S_+ + S_+\left(P^\dagger_q a_{p-}a_{p+} - a^\dagger_{q-}a^\dagger_{q+}P_p\right) + S_-\left(P^\dagger_q a_{p-}a_{p+} - a^\dagger_{q-}a^\dagger_{q+}P_p\right)\right) \tag{28}$$

Substituting in P_p we have

$$-g\textstyle\sum_{pq}\left(\left(P^\dagger_q P_p - P^\dagger_q P_p\right)S_- + \left(P^\dagger_q P_p - P^\dagger_q P_p\right)S_+ + S_+\left(P^\dagger_q P_p - P^\dagger_q P_p\right) + S_-\left(P^\dagger_q P_p - P^\dagger_q P_p\right)\right) = 0 \tag{29}$$

1.2 Hamiltonian Commutes with $P_p^+ P_p^-$

In order to show it keeps pairs together we must show that it conserves the product of the pair creation and annhilation operators.

Now we can then take advantage of some basic commutator algebra to solve that

$$\left[P_p^\dagger P_p, P_q^\dagger P_r\right] = P_p^\dagger \left[P_p, P_q^\dagger\right] P_r + P_q^\dagger P_q^\dagger \left[P_p, P_r\right] + \left[P_p^\dagger, P_q^\dagger\right] P_r P_p + P_q^\dagger \left[P_p^\dagger, P_r\right] P_p \quad (30)$$

$$= \delta_{p,q} P_p^\dagger P_r - \delta_{p,r} P_q^\dagger P_p = P_q^\dagger P_r - P_q^\dagger P_r = 0 \quad (31)$$

$$\left[P_p^\dagger P_p, a_{r\sigma}^\dagger a_{r\sigma}\right] = P_p^\dagger \left[P_p, a_{r\sigma}^\dagger\right] a_{r\sigma} + P_p^\dagger a_{r\sigma}^\dagger \left[P_p, a_{r\sigma}\right] + \left[P_p^\dagger, a_{r\sigma}^\dagger\right] a_{r\sigma} P_p + a_{r\sigma}^\dagger \left[P_p^\dagger, a_{r\sigma}\right] P_p \quad (32)$$

$$\left[P_p^\dagger P_p, a_{r\sigma}^\dagger a_{r\sigma}\right] = P_p^\dagger \delta_{p,r} a_{p-\sigma} a_{r\sigma} - \delta_{p,r} a_{r\sigma}^\dagger a_{p-\sigma}^\dagger P_p = P_p^\dagger P_p - P_p^\dagger P_p = 0 \quad (33)$$

2 Hilbert Space of Two Lowest Single Particle States, No Broken Pairs

We can then establish first that $S = 0$ and $S_z = 0$. It thereby follows that

$$\hat{H} = a_{2+}^\dagger a_{2+} + a_{2-}^\dagger a_{2-} - g\left(P_1^\dagger P_1 + P_1^\dagger P_2 + P_2^\dagger P_1 + P_2^\dagger P_2\right) \quad (34)$$

Now let us consider the case of the two eigenstates represented as

$$\psi = \begin{pmatrix} \psi_1 \\ \psi_2 \end{pmatrix} \quad (35)$$

Where the ψ_i represents the states where a fermion pair is in state 1 or state 2. Then we have that

$$\hat{H} = \begin{pmatrix} -g & -g \\ -g & -g+2 \end{pmatrix} \quad (36)$$

We see that this can be exactly solved for eigenvalues of

$$(-g-\lambda)(-g+2-\lambda) - g^2 = \lambda^2 + 2g\lambda - 2g - 2\lambda = 0 \quad (37)$$

So we have

$$\lambda = -g + 1 \pm \sqrt{(g-1)^2 + 2g} \quad (38)$$

3 Four Particles paired in p=1,2,3,4 states

Here we see that a similar matrix can be constructed easily from the slater determinants.

Let us denote a state with pairs in states p=1, p=2 as $(1,2)$. Then we order these states as

$$[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)] \tag{39}$$

We then construct the matrix in this order as(noting there are two combinations which will lead to the same bra and ket picking up a g term, and those where there are no overlaps will have none)

$$\begin{pmatrix} 2-2g & -g & -g & -g & -g & 0 \\ -g & 4-2g & -g & -g & 0 & -g \\ -g & -g & 6-2g & 0 & -g & -g \\ -g & -g & 0 & 6-2g & -g & -g \\ -g & 0 & -g & -g & 8-2g & -g \\ 0 & -g & -g & -g & -g & 10-2g \end{pmatrix} \tag{40}$$

We can now find the eigenvalues only numerically.

4 Code For Setting Up Matricies and Obtaining Eigenvalues

In order to solve these problems, I have written a code which does a few things: First, it takes an input basis of states which can be occupied. While used here for the base case of p,σ, it could be extended to any arbitrary quantum numbers. These can then be put into a hilbert space object, which orders and keeps track of the states, as well as generates the maximal Hilbert Space spanning this basis.

In order to restrict the basis a number of specific functions were also made, which determine if the sum of sigmas is 0 or if all states are paired.

At the heart of the machinery then are the creation and annhilation operators, which are constructed for each state. When applied to a Slater Determinant object, these return either 0 or a modified state vector (depending on what is appropriate.) With the help of an auxiliary function sort from the hilbert space, it is possible to then keep track of the phase as well. Finally, a Hamiltonian class incorporates these operators into one object which can then select a bra and ket to determine a matrix element from. This allows for the simple construction of a numpy matrix H for a given hilbert space(based on number of available quantum numbers, number of particles, pairing, etc.). This can easily be diagonalized using standard numpy algorithms, from which eigenvalues can be determined. In this case the single particle states were developed directly in the program. However, a generalization of the program to read in states from an input file

would be trivial using numpy.genfromtxt.

In order to compare the results, I show below the resulting eigenvalues generated from the 4 p states 4 particle states and 6 ps states 6 particle states.

For g=0 we see immediately we have in the 4p state case

$$E = [2, 8, 10, 6, 6, 4] \tag{41}$$

We see these match exactly with the values from the above matrix(which is diagonal with g=0).

We can also consider the degenerate case where we allow the energy of all states to be 1. This leads to in the case of the 4 particles and 4 p states to a diagonal matrix $4I$, which just counts the particles in each state. Allowing now g=1 and H_0 to be 0 we have eigenvalues of

$$E = [-6, -2, -2, 0, 0] \tag{42}$$

We know from the base case of degeneracy that as the degeneracy is 4 in p and 2 in spin and so totals to 8 states, and such we should expect a ground state of

$$E_0 = -\frac{g}{4} n \left(\Omega - n + 2\right) = -\left(8 - 4 + 2\right) = -6 \tag{43}$$

We see the program's results agree with the prediction.

5 Varying g

For the larger scale cases the program takes more time to run. To that end I designed a function to run through a number of values for g in a given hilbert space, forming Hamiltonians, and from there calculating the matricies, finding their eigenvalues and graphing each of these on a scatterplot vs g. The results of this are shown below.

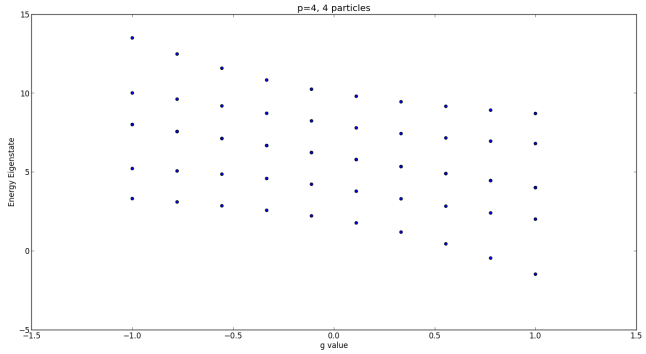

Figure 1: Effect of changing g on energy eigenstates for 4 particles and states

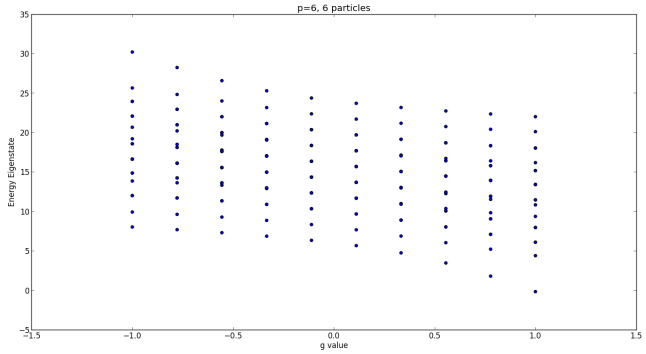

Figure 2: Effect of changing g on energy eigenstates for 6 particles and states

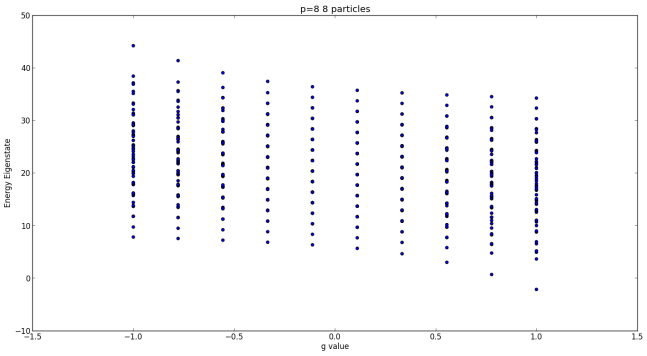

Figure 3: Effect of changing g on energy eigenstates for 8 particles and states

We note that while the proliferation of energy states with a larger number of levels and states is a striking feature, it is of course expected. What we may note of more

interest, perhaps, is that there is a lifting of degeneracies as g becomes larger, for example there are 10 clearly visible states near g=0 for six particles and states, but 13 energy states near g=1. A similar pattern is clear for the 8 particles and states. We also note that the energy shifting is not equal among all energies, but that for large g the ground state is shifted much lower than the highest state, compared to g=0, and similarly at $g \approx -1$ the reverse holds true, with the highest state showing a vast shift upwards.

6 Code

6.1 Module

```
import itertools
import numpy as np
from itertools import combinations as cwr
import matplotlib.pyplot as plt
import copy


class Hamiltonian:
    Terms=[]
    def braket(self,bra,ket):
        i=0
        for term in self.Terms:
            i+=term.braket(bra,ket)
        return i
    def Act(self,state):
        if type(state) is statesum:
            ans=[]
            for s in statesum.states:
                ans1=(self.Act(s))
                for item in ans1:
                    ans.append(item)
            return statesum(ans)
        if type(state) is SlaterDeterm:
            ans=[]
            for term in self.Terms:
                ans.append(term.Act(state)[0])
            return statesum(ans)
    def __init__(self,Terms):
        self.Terms=Terms
class statesum:
    State=[]
    def dot(self,state):
```

```
        ans=[]
        for s in self.states:
            ans.append(s.dot(state))
        print(ans)
        total=0
        for i in ans:
            total+=i
        return total
    def __init__(self,states):
        for s in states:
            if type(s) is not SlaterDeterm:
                raise NameError('statesum defined incorrectly')
        self.State=states
class HilbertSpace:
    States=[]
    Slaters=[]
    def Copy(self):
        statecopy=[]
        Slaterscopy=[]
        for s in self.States:
            statecopy.append(s)
        for s in self.Slaters:
            Slaterscopy.append(s)
        hcopy=HilbertSpace([])
        hcopy.States=statecopy
        hcopy.Slaters=Slaterscopy
        return hcopy
    def GetStateIndex(self,state):
        for i,st in enumerate(self.States):
            if st.Compare(state):
                return i
        raise NameError('State Not in Hilbert Space')
    def Sort(self,slater):
        if type(slater) is not SlaterDeterm:
          raise NameError('Wrong Input To Get Phase')
        if slater.State==0:
            return [slater,0]
        indicies=[]
        if slater.AreStatesRepeated():
            sv=slater.copy()
            sv.State=0
            return [sv,0]
        for state in slater.State:
```

```
            # print(indicies)
              indicies.append(self.GetStateIndex(state))
        sortslate=sorted(indicies)
        sortedslate=slater.State.copy()
        for i,j in enumerate(sortslate):
            sortedslate[i]=self.States[j]
        perms=0
        while indicies!=sortslate:
            for i in range(len(indicies)):
                if indicies[i]!=sortslate[i]:
                    for j in range(len(indicies)):
                        if indicies[j]==sortslate[i]:
                            indicies[j]=indicies[i]
                            indicies[i]=sortslate[i]
                            perms=perms+1
                            break
        return [SlaterDeterm(sortedslate),perms]
    def GenerateSlaters(self):
        for i in range(len(self.States)+1):
            for comb in cwr(self.States,i):
                slate=SlaterDeterm(list(comb))
                self.Slaters.append(self.Sort(slate)[0])
    def __init__(self,States):
        self.States=States
        self.Slaters=[]
        self.GenerateSlaters()



class HamilTerm:
    constant=1
    Operators=[]
    hilb=None
    def Act(self,statevec):
        phase=1
        sv=statevec.copy()
        if len(self.Operators)==0:
            return sv
        for op in self.Operators:
            phase*=op.GetPhase(self.hilb,sv)
            sv=op.Act(sv)
        sv1,phase1=self.hilb.Sort(sv)
        #     phase=phase*phase1
```

```
            #   print(sv.State)
            #if self.hilb is not None:
             #    sv, phase1=self.hilb.Sort(sv)
            #self.constant=self.constant*(-1)**phase
            return [sv1,phase]
        def braket(self,bra,ket):
            sv,phase=self.Act(ket)
          #  phase=2
            #print(sv)
            #print(phase)
            #print(self.constant)
#           print(self.constant*bra.dot(sv)*(-1)**phase)
 #          print(self.constant*bra.dot(ket)*(-1)**phase)
            #return self.constant*bra.dot(sv)*(-1)**(phase)
            return self.constant*bra.dot(sv)*(phase)
        def __init__(self,ops,hilb=None,constant=1):
            self.Operators=ops
            self.hilb=hilb
            self.constant=constant
    class SlaterDeterm:
        State=[]
        def copy(self):
            copystate=[]
            try:
                for s in self.State:
                    copystate.append(s.copy())
            except:
                    copystate=0
            return SlaterDeterm(copystate)
        def AreStatesRepeated(self):
            for i, state in enumerate(self.State):
                for j,state1 in enumerate(self.State):
                    if i!=j and state1.Compare(state):
                        self.State=0
                        return True
            return False
        def Compare(self,slat):
            if type(slat) is statesum:
                ans=[]
                for st in statesum.State:
                   # print(st)
                    ans.append(self.Compare(st))
                return ans
```

```
        if type(slat) is not SlaterDeterm:
            # print(type(slat))
            #  print(slat)
            raise NameError('Not Slater Determinant in Comparison')
        if slat.State==0:
            if self.State==0:
                return True
            return False
        if self.State==0:
            return False
        if len(slat.State) != len(self.State):
            return False
        for i in range(len(self.State)):
            if not self.State[i].Compare(slat.State[i]):
                return False
        return True
    def __init__(self,states):
        if states==0:
            self.State=0
            return None
        for a in states:
            if type(a) is not State:
                raise NameError('Not a state in Slater Determinent')
        self.State=states
        if self.AreStatesRepeated():
            self.State=0
    def dot(self,slat):
        if type(slat)is statesum:
            ans=[]
            #print(ans)
            for item in slat.State:
             #    print(item)
                ans.append(self.dot(item))
            total=0
            for i in ans:
                total+=i
            return total
        if self.State==0 or slat.State==0:
            return 0
        if self.Compare(slat):
            return 1
        return 0
```

```
class QuantumNumber:
    name=""
    value=0
    def __init__(self,name,value):
        self.name=name
        self.value=value
    def copy(self):
        return QuantumNumber(self.name,self.value)
    def compare(self,qn):
        if qn.name==self.name:
            if qn.value==self.value:
                return True
        return False
class Operator:
    state=0
    Creation=True
    def GetPhase(self,Hilb,Slate):
        index=-1
        nbetween=0
        for i,st in enumerate(Hilb.States):
            if self.state.Compare(st):
                index=i
                break
        if Slate.State==[] or Slate.State==0:
            return 0
        if index==-1:
            raise NameError('State not in HilbertSpace')
        for state in Slate.State:
            if Hilb.GetStateIndex(state)<index:
                nbetween+=1
        return (-1)**nbetween
    def __init__(self,state,Creation):
        self.state=state
        self.Creation=Creation
    def copy(self):
        return Operator(self.state.copy(),self.Creation)
    def Act(self,statevector):
        if statevector.State==0:
            return statevector.copy()
        if self.Creation:
            for state in statevector.State:
```

```
                if self.state.Compare(state):
                    sv=statevector.copy()
                    sv.State=0
                    return sv
            sv=statevector.copy()
            sv.State.append(self.state)
            return sv
        else:
            for state in statevector.State:
                if self.state.Compare(state):
                    sv=statevector.copy()
                    for s in sv.State:
                        if s.Compare(state):
                            sv.State.remove(s)
                            return sv
            sv=statevector.copy()
            sv.State=0
            return sv

class State:
    quantumnums=[]
    anop=0
    creop=0
    def copy(self):
        qcopy=[]
        for q in self.quantumnums:
            qcopy.append(q.copy())
        return State(qcopy)
    def __init__(self,quantumnums):
        self.quantumnums=quantumnums
        self.creop=Operator(self,True)
        self.anop=Operator(self,False)
        if quantumnums==[]:
            self.quantumnums=0
    def Compare(self,state):
        if len(self.quantumnums) != len(state.quantumnums):
            return False
        for q in range(len(self.quantumnums)):
            if not self.quantumnums[q].compare(state.quantumnums[q]):
                return False
        return True

        return True
```

```
    def GetVal(self,name):
        for q in self.quantumnums:
            if q.name==name:
                return q.value



def MakeHSpaceM(hilbert,M=0):
    slates=[]
    hcop=hilbert.Copy()
    for state in hcop.Slaters:
        i=0
        for s in state.State:
            i+=s.GetVal('sigma')
        if i==M:
            slates.append(state)
    hcop.Slaters=slates
    return hcop
def PairingH(hilbert):
     slates=[]
     hcop=hilbert.Copy()
     for slate in hcop.Slaters:
        i=0
        if slate.State==0 or slate.State==[]:
             slates.append(slate)
        else:
            for state in slate.State:
                #con=True
                for state1 in slate.State:
                    if state.GetVal('sigma') != state1.GetVal('sigma') and sta
                        #slates.append(slate)
                        i=i+1
            if i==len(slate.State):
                slates.append(slate)
                 #        con=False

     hcop.Slaters=list(set(slates))
     return hcop
def MakeNumParticles(hilbert,n):
    slates=[]
    hcop=hilbert.Copy()
    for state in hcop.Slaters:
        if state.State==0:
```

```
            slates.append(state)
        else:
            if len(state.State)==n:
                slates.append(state)
    hcop.Slaters=slates
    return hcop
def MakeMatrix(HSpace,H):
    n=len(HSpace.Slaters)
    mat=np.zeros([n,n])
    for i, j in itertools.product(range(n),range(n)):
        mat[i][j]=H.braket(HSpace.Slaters[i],HSpace.Slaters[j])
#           # print(mat[i][j])
    return mat

#def ReadInMatrixElements(fil):
 #    f=np.genfromtxt(open(fil))
```

6.2 Implementation

```
from ShellModel import *
import itertools
import matplotlib.pyplot as plt
pstates=[]
numpstates=8
for i in range(numpstates):
    pstates.append(QuantumNumber('p',i+1))
sigmastates=[]
for i in [-1,1]:
    sigmastates.append(QuantumNumber('sigma',i))
#We construct the HilbertSpaces for all the cases under consideration

#SingleParticleStates4=[]
#for item in itertools.product(pstates[0:4],sigmastates):
 #    SingleParticleStates4.append(State(item))
#Hilb4pstates2particles=PairingH(MakeNumParticles(MakeHSpaceM(HilbertSpace(Si



#SingleParticleStates2=[]
#for item in itertools.product(pstates[0:2],sigmastates):
 #    SingleParticleStates2.append(State(item))
#Hilb2pstates2particles=PairingH(MakeNumParticles(MakeHSpaceM(HilbertSpace(Si

#SingleParticleStates6=[]
#for item in itertools.product(pstates[0:6],sigmastates):
```

```
    #SingleParticleStates6.append(State(item))
#Hilb6pstates2particles=MakeNumParticles(MakeHSpaceM(PairingH(HilbertSpace(Si
#SingleParticleStates8=[]
#for item in itertools.product(pstates[0:8],sigmastates):
    #SingleParticleStates8.append(State(item))
#Hilb8pstates2particles=(MakeHSpaceM(PairingH(HilbertSpace(SingleParticleState
#Hilb8pstates2particles=MakeNumParticles(Hilb8pstates2particles,4)

#Next we want to construct the Hamiltonian terms


def MakeStates(pstate,sstate,p):
    states=[]
    for i in itertools.product(pstate[0:p],sstate):
        states.append(State(i))
    return states

def MakeHilbertSpace(States,Numparts,Pairing=True,MState=0, M=True):
    Hilb=HilbertSpace(States)
    Hilb=MakeNumParticles(Hilb,Numparts)
    if(M):
        Hilb=MakeHSpaceM(Hilb,MState)
    if Pairing:
        Hilb=PairingH(Hilb)
    return Hilb
def MakeH0Term(HSpace):
    terms=[]
    for s in HSpace.States:
        ops=[0,0]
        ops[0]=s.anop
        ops[1]=s.creop
        terms.append(HamilTerm(ops,HSpace,s.GetVal('p')-1))
    return terms
def MakeDegenH0Term(HSpace,const):
    terms=[]
    for s in HSpace.States:
        ops=[0,0]
        ops[0]=s.anop
        ops[1]=s.creop
        terms.append(HamilTerm(ops,HSpace,const))
    return terms

def MakeH1Term(HSpace,Constant):
```

```
    ps=[]
    for state in HSpace.States:
        ps.append(state.GetVal('p'))
    ps=list(set(ps))
    terms=[]
    for p,q in itertools.product(ps,ps):
        ops=[0,0,0,0]
        for state in HSpace.States:
            if state.GetVal('p')==q and state.GetVal('sigma')==1:
              ops[0]=(state.anop)
            if state.GetVal('p')==q and state.GetVal('sigma')==-1:
                ops[1]=state.anop
            if state.GetVal('p')==p and state.GetVal('sigma')==1:
              ops[3]=(state.creop)
            if state.GetVal('p')==p and state.GetVal('sigma')==-1:
                ops[2]=state.creop
        terms.append(HamilTerm(ops,HSpace,Constant))
    return terms
#def MakeMatrix(HSpace,H):
 #   n=len(HSpace.Slaters)
  #  mat=np.zeros([n,n])
   # for i, j in itertools.product(range(n),range(n)):
    #      mat[i][j]=H.braket(HSpace.Slaters[i],HSpace.Slaters[j])
#          # print(mat[i][j])
    #return mat

def MakeProblemHamil(hilb,g):
    return Hamiltonian(MakeH0Term(hilb)+MakeH1Term(hilb,-g))
def GetEigenEns(mat):
    return np.linalg.eig(mat)[0]


def MakePlot(gs,h1,title=""):
    plots=[]
    for g in gs:
        H=MakeProblemHamil(h1,g)
        m1=MakeMatrix(h1,H)
        en=GetEigenEns(m1)
        for e in en:
            plots.append([g,e])
    plots=np.array(plots)
    plt.scatter(plots[:,0],plots[:,1])
    plt.title(title)
```

```
    plt.xlabel('g value')
    plt.ylabel('Energy Eigenstate')
plt.show()

gs=np.linspace(-1,1,10)
#g=1
s1=MakeStates(pstates,sigmastates,4)
h0=MakeHilbertSpace(s1,4)
s2=MakeStates(pstates,sigmastates,6)
h2=MakeHilbertSpace(s2,6)
s3=MakeStates(pstates,sigmastates,8)
h3=MakeHilbertSpace(s3,6)
#MakePlot(gs,h0,'p=4, 4 particles')
#MakePlot(gs,h2,'p=6, 6 particles')
MakePlot(gs,h3,'p=8 8 particles')
#H1=MakeProblemHamil(h1,g)
#m1=MakeMatrix(h1,H1)
#print(m1)
def MakePlot(gs,h1,title=""):
    plots=[]
    for g in gs:
        H=MakeProblemHamil(h1,g)
        m1=MakeMatrix(h1,H)
        en=GetEigenEns(m1)
        print(g)
        for e in en:
            plots.append([g,e])
    plots=np.array(plots)
    a=plt.axes()
    a.scatter(plots[:,0],plots[:,1])
    a.title(title)
    a.xlabel('g value')
    a.ylabel('Energy Eigenstate')
    a.show()
  #  plt.show()

#g=-1
#H0=Hamiltonian(MakeH0Term(Hilb2pstates2particles)+MakeH1Term(Hilb2pstates2pa

#H1=Hamiltonian(MakeH0Term(Hilb4pstates2particles)+MakeH1Term(Hilb4pstates2pa
#H2=Hamiltonian(MakeH0Term(Hilb6pstates2particles)+MakeH1Term(Hilb6pstates2pa
#3=Hamiltonian(MakeH0Term(Hilb8pstates2particles)+MakeH1Term(Hilb8pstates2par
#Mat0=MakeMatrix(Hilb2pstates2particles,H0)
```

```
#Mat1=MakeMatrix(Hilb4pstates2particles,H1)
#Mat2=MakeMatrix(Hilb6pstates2particles,H2)
#Mat3=MakeMatrix(Hilb8pstates2particles,H3)
#print(np.linalg.eig(Mat1))
#print(np.linalg.eig(Mat2))
#print(np.linalg.eig(Mat3))
#print(Mat0)
#print(Mat1)
#print(Mat2)
#print(Mat3)
```