Slater determinants as basis states, Repetition

The simplest possible choice for many-body wavefunctions are product wavefunctions. That is

\[ \psi(x_1, x_2, \ldots, x_N) \approx \phi(x_1)\phi(x_2)\phi(x_3) \ldots \]

because we are really only good at thinking about one particle at a time. Such product wavefunctions, without correlations, are easy to work with; for example, if the single-particle states \( \phi_i(x) \) are orthonormal, then the product wavefunctions are easy to orthonormalize.

Similarly, computing matrix elements of operators are relatively easy, because the integrals factorize. The price we pay is the lack of correlations, which we must build up by using many, many product wavefunctions. (Thus we have a trade-off: compact representation of correlations but difficult integrals versus easy integrals but many states required.)

Slater determinants as basis states, Repetition

Because we have fermions, we are required to have antisymmetric wavefunctions, e.g.

\[ \psi(x_1, x_2, x_3, \ldots, x_N) = - \psi(x_2, x_1, x_3, \ldots, x_N) \]

e etc. This is accomplished formally by using the determinantal formalism

\[ \psi(x_1, x_2, \ldots, x_N) = \frac{1}{\sqrt{N!}} \det \begin{vmatrix} \phi(x_1) & \phi(x_2) & \cdots & \phi(x_N) \\ \phi(x_1) & \phi(x_2) & \cdots & \phi(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ \phi(x_1) & \phi(x_2) & \cdots & \phi(x_N) \end{vmatrix} \]

Product wavefunction + antisymmetry = Slater determinant.

Slater determinants as basis states

As a practical matter, however, Slater determinants beyond \( N = 4 \) quickly become unwieldy. Thus we turn to the occupation representation or second quantization to simplify calculations.

The occupation representation, using fermion creation and annihilation operators, is compact and efficient. It is also abstract and, at first encounter, not easy to internalize. It is inspired by other operator formalism, such as the ladder operators for the harmonic oscillator or for angular momentum, but unlike those cases, the operators do not have coordinate space representations.

Instead, one can think of fermion creation/annihilation operators as a game of symbols that compactly reproduces what one would do, albeit clumsily, with full coordinate-space Slater determinants.

Quick repetition of the occupation representation

We start with a set of orthonormal single-particle states \( \{ \phi_i(x) \} \).

(Note: this requirement, and others, can be relaxed, but leads to a more involved formalism.) Any orthonormal set will do. To each single-particle state \( \phi_i(x) \) we associate a creation operator \( \hat{a}_i^\dagger \) and an annihilation operator \( \hat{a}_i \).

When acting on the vacuum state \( |0\rangle \), the creation operator \( \hat{a}_i^\dagger \) causes a particle to occupy the single-particle state \( \phi_i(x) \):

\[ \phi_i(x) \rightarrow \hat{a}_i^\dagger |0\rangle \]

Properties of the determinant (interchange of any two rows or any two columns yields a change in sign; thus no two rows and no two columns can be the same) lead to the Pauli principle:

- No two particles can be in the same state (two rows the same);
- No two particles can be in the same state (two columns the same).
But with multiple creation operators we can occupy multiple states:
$$\phi(x)\phi(x')\phi(x'') \rightarrow \hat{a}[a][a'][a'']|0\rangle.$$  
Now we impose antisymmetry, by having the fermion operators satisfy anticommutation relations:
$$\hat{a}[a][a'] - \hat{a}[a'][a] = 0$$  
so that
$$\hat{a}[a][a'] = -\hat{a}[a'][a].$$

Quick repetition of the occupation representation

Because of this property, automatically $\hat{a}[a][a] = 0$, enforcing the Pauli exclusion principle. Thus when writing a Slater determinant using creation operators,
$$|\Phi_n\rangle = \prod_{i<j} \hat{a}_i^\dagger|0\rangle,$$
each index $i, j, k, \ldots$ must be unique.

For some relevant exercises with solutions see chapter 8 of Lecture Notes in Physics, volume 936.

Full Configuration Interaction Theory

We have defined the ansatz for the ground state as
$$|\Phi_0\rangle = \prod_{i<j} \hat{a}_i^\dagger|0\rangle,$$
where the index $i$ defines different single-particle states up to the Fermi level. We have assumed that we have $N$ fermions. A given one-particle-one-hole $(1p1h)$ state can be written as
$$|\Phi_{1p1h}\rangle = \hat{a}_i|0\rangle\hat{a}_j|0\rangle|\Phi_0\rangle,$$
while a $2p2h$ state can be written as
$$|\Phi_{2p2h}\rangle = \hat{a}_i\hat{a}_j|0\rangle\hat{a}_k\hat{a}_l|\Phi_0\rangle,$$
and a general $NpNh$ state as
$$|\Phi_{NpNh}\rangle = \hat{a}_1\hat{a}_2\ldots\hat{a}_N|\Phi_0\rangle.$$  

Quick repetition of the occupation representation

We can then expand our exact state function for the ground state as
$$|\Psi_0\rangle = C_0|\Phi_0\rangle + \sum_{ai} C_{ai}|\Phi_{ai}\rangle + \sum_{abij} C_{abij}|\Phi_{abij}\rangle + \ldots$$
in a more compact form as
$$|\Psi_0\rangle = \sum_{i\neq j} C_i^p C_j^h|\Phi_{1p1h}\rangle + \sum_{i\neq j\neq k} C_i^p C_j^h C_k^h|\Phi_{2p2h}\rangle + \ldots$$
where $N$ stands for $0, 1, \ldots, n$ hole states and $P$ for $0, 1, \ldots, n$ particle states. Our requirement of unit normalization gives
$$\langle \Psi_0|\Psi_0\rangle = \sum_{i\neq j} |C_i|^2 = 1,$$
and the energy can be written as
$$E = \langle \Psi_0|\hat{H}|\Psi_0\rangle = \sum_{i\neq j} (C_i^p)^* C_i^p + (C_j^h)^* C_j^h + \sum_{i\neq j\neq k} (C_i^p)^* C_j^h C_k^h.$$  
Normally
$$E = \langle \Psi_0|\hat{H}|\Psi_0\rangle = \sum_{P \neq \top} C_P^p C_P^h |\Phi_P\rangle |\Phi_P\rangle^*,$$
is solved by diagonalization setting up the Hamiltonian matrix defined by the basis of all possible Slater determinants. A diagonalization is equivalent to finding the variational minimum of
$$\langle \Psi_0|\hat{H}|\Psi_0\rangle - \lambda \langle \Psi_0|\Psi_0\rangle,$$
where $\lambda$ is a variational multiplier to be identified with the energy of the system. The minimization process results in
$$\delta \left( \langle \Psi_0|\hat{H}|\Psi_0\rangle - \lambda \langle \Psi_0|\Psi_0\rangle \right) = 0,$$
$$\sum_{P \neq \top} \left( \delta C_P^p C_P^h |\Phi_P\rangle |\Phi_P\rangle^* \right) C_P^p C_P^h + \delta C_P^p C_P^h |\Phi_P\rangle |\Phi_P\rangle^* C_P^p C_P^h = 0 - \lambda \delta C_P^p C_P^h.$$  
Since the coefficients $(C_P^p)$ and $(C_P^h)$ are complex conjugates it
Full Configuration Interaction Theory

This leads to
\[ \sum_{P \in \mathcal{P}} \sum_{PH \in \mathcal{PH}} \left[ \Phi_P^H | \hat{H} | \Phi_{P'}^H \right] C_{P'H}^H - \lambda \sum_{P \in \mathcal{P}} \sum_{PH \in \mathcal{PH}} \left[ \Phi_P^H | \hat{H} | \Phi_{P'}^H \right] C_{P'H}^H = 0, \]
for all sets of \( P \) and \( H \).
If we then multiply by the corresponding \( C_{P'H}^H \) and sum over \( PH \) we obtain
\[ \sum_{P \in \mathcal{P}} \sum_{PH \in \mathcal{PH}} \left[ \Phi_P^H | \hat{H} | \Phi_{P'}^H \right] C_{P'H}^H - \lambda \sum_{P \in \mathcal{P}} \sum_{PH \in \mathcal{PH}} \left[ \Phi_P^H | \hat{H} | \Phi_{P'}^H \right] C_{P'H}^H = 0, \]
leading to the identification \( \lambda = E \). This means that we have for all \( PH \) sets
\[ \sum_{P \in \mathcal{P}} \left[ \Phi_P^H | \hat{H} - E | \Phi_{P'}^H \right] = 0. \tag{1} \]

Example of a Hamiltonian matrix

Suppose, as an example, that we have six fermions below the Fermi level. This means that we can make at most \( 6p - 6h \) excitations. If we have an infinity of single particle states above the Fermi level, we will obviously have an infinity of say \( 2p - 2h \) excitations. Each such way to configure the particles is called a configuration. We will always have to truncate in the basis of single-particle states. This gives us a finite number of possible Slater determinants. Our Hamiltonian matrix would then look like (where each block can have a large dimensionality):

<table>
<thead>
<tr>
<th>( 1p - 6h )</th>
<th>( 1p - 1h )</th>
<th>( 2p - 2h )</th>
<th>( 3p - 3h )</th>
<th>( 4p - 4h )</th>
<th>( 5p - 5h )</th>
<th>( 6p - 6h )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1p - 1h )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( 2p - 2h )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( 3p - 3h )</td>
<td>( 0 )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( 4p - 4h )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( 5p - 5h )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
<tr>
<td>( 6p - 6h )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
<td>( x )</td>
</tr>
</tbody>
</table>

with a two-body force. Why are there non-zero blocks of elements?

Shell-model jargon

If we do not make any truncations in the possible sets of Slater determinants (many-body states) we can make by distributing \( A \) nucleons among \( n \) single-particle states, we call such a calculation for Full configuration interaction theory
If we make truncations, we have different possibilities
- The standard nuclear shell-model. Here we define an effective Hilbert space with respect to a given core. The calculations are normally then performed for all many-body states that can be constructed from the effective Hilbert spaces. This approach requires a properly defined effective Hamiltonian
- We can truncate in the number of excitations. For example, we can limit the possible Slater determinants to only \( 1p - 1h \) and \( 2p - 2h \) excitations. This is called a configuration interaction calculation at the level of singles and doubles excitations, or just CISD.
- We can limit the number of excitations in terms of the excitation energies. If we do not define a core, this defines the basis, this is called the restricted basis model approach

FCI and the exponential growth

Full configuration interaction theory calculations provide in principle, if we can diagonalize numerically, all states of interest. The dimensionality of the problem explodes however quickly.

The total number of Slater determinants which can be built with \( N \) neutrons distributed among \( n \) single particle states is
\[ \frac{n!}{(n-N)!}. \]

For a model space which comprises the first for major shells only \( 0s, 0p, 1s0d \) and \( 1s0d \) we have 40 single particle states for neutrons and protons. For the eight neutrons of oxygen-16 we would then have
\[ \frac{16!}{8!} \sim 10^{10}, \]
and multiplying this with the number of proton Slater determinants we end up with approximately with a dimensionality of \( d \sim 10^{14} \).
Exponential wall

This number can be reduced if we look at specific symmetries only. However, the dimensionality explodes quickly!

- For Hamiltonian matrices of dimensionality which are smaller than $d \sim 10^5$, we would use so-called direct methods for diagonalizing the Hamiltonian matrix.
- For larger dimensionality iterative eigenvalue solvers like Lanczos’ method are used. The most efficient codes at present can handle matrices of $d \sim 10^{10}$.

Rewriting the FCI equation

In our notes on Hartree-Fock calculations, we have already computed the matrix $\langle \Phi_0 | \hat{H} | \Phi_a \rangle$ for the Hartree-Fock basis. If we are using a Hartree-Fock basis, then the matrix elements $\langle \Phi_a | \hat{H} | \Phi_b \rangle$ are zero and we are left with a correlation energy given by

$$E - E_0 = \Delta E_{\text{HF}} = \sum_{ab} \langle \Phi_0 | \hat{H} | \Phi_a \rangle \langle \Phi_a | \hat{H} | \Phi_b \rangle C_{ab}^{ij}$$

A non-practical way of solving the eigenvalue problem

To see this, we look at the contributions arising from

$$\langle \Phi_0 | \hat{H} | \Phi_a \rangle - \langle \Phi_0 | \hat{H} | \Phi_0 \rangle$$

in Eq. (1), that is we multiply with $\langle \Phi_0 |$ from the left in

$$(\hat{H} - E) \sum_{P} C^{|P\rangle}_{P^\prime} \langle \Phi_0 | P^\prime \rangle = 0.$$
Rewriting the FCI equation, does not stop here

We need more equations. Our next step is to set up
\[
\langle \Phi_i | H - E \Phi_i \rangle + \sum_{jk} \langle \Phi_i | H - E \Phi_j \rangle C_{ij} + \sum_{kl} \langle \Phi_i | H - E \Phi_k \rangle C_{ik} + \sum_{jklm} \langle \Phi_i | H - E \Phi_{jkl} \rangle C_{ijkl} + \sum_{klnm} \langle \Phi_i | H - E \Phi_{km} \rangle C_{iknm} = \langle \Phi_i | \hat{f} | a \rangle \epsilon_i - \epsilon_a.
\]

Definition of the correlation energy

The correlation energy is defined as, with a two-body Hamiltonian,
\[
\Delta E = \sum_{ab} \langle i | \hat{v} | ab \rangle C_{ab} + \sum_{AM} \langle i | \hat{a} | b \rangle C_{ab}.
\]

Rewriting the FCI equation, more to add

The observant reader will however see that we need an equation for
\[C_{bc}^{jk}\] and \[C_{cd}^{kl}\] as well. To find equations for these coefficients we need then to continue our multiplications from the left with the various \[\Phi_{ab}^{ij}\] terms.

For \[C_{bc}^{jk}\] we need then
\[
\langle \Phi_{bc}^{jk} | H - E \Phi_{bc}^{jk} \rangle C_{bc}^{jk} + \sum_{abcd} \langle \Phi_{bc}^{jk} | H - E \Phi_{abcd} \rangle C_{bcd}^{kl} + \sum_{jk} \langle \Phi_{bc}^{jk} | H - E \Phi_{jk} \rangle C_{jk} = \langle \Phi_{bc}^{jk} | \hat{v} | ab \rangle \epsilon_i + \langle \Phi_{bc}^{jk} | \hat{f} | a \rangle \epsilon_i - \epsilon_a.
\]

We can compute easily expectation values of other operators, as well as transition probabilities.

Correlations are easy to understand in terms of contributions to a given operator beyond the Hartree-Fock contribution.

This is the standard approach in many-body theory.

Summarizing FCI and bringing in approximative methods

If we can diagonalize large matrices, FCI is the method of choice since:
\begin{itemize}
  \item It gives all eigenvalues, ground state and excited states
  \item The eigenvectors are obtained directly from the coefficients \[C_{ab}^{ij}\] which result from the diagonalization
  \item We can compute easily expectation values of other operators, as well as transition probabilities
  \item Correlations are easy to understand in terms of contributions to a given operator beyond the Hartree-Fock contribution
\end{itemize}

This is the standard approach in many-body theory.

Definition of the correlation energy

The correlation energy is defined as, with a two-body Hamiltonian,
\[
\Delta E = \sum_{ab} \langle i | \hat{v} | ab \rangle C_{ab} + \sum_{AM} \langle i | \hat{a} | b \rangle C_{ab}.
\]

The coefficients \[C\] result from the solution of the eigenvalue problem. The energy of say the ground state is then
\[
E = E_{\text{HF}} + \Delta E,
\]
where the so-called reference energy is the energy we obtain from a Hartree-Fock calculation, that is
\[
E_{\text{HF}} = \langle \Phi_0 | H | \Phi_0 \rangle.
\]
FCI equation and the coefficients

However, as we have seen, even for a small case like the four first major shells and a nucleus like oxygen-16, the dimensionality becomes quickly intractable. If we wish to include single-particle states that reflect weakly bound systems, we need a much larger single-particle basis. We need thus approximative methods that sum specific correlations to infinite order. Popular methods are:
- Many-body perturbation theory (in essence a Taylor expansion)
- Coupled cluster theory (coupled non-linear equations)
- Green’s function approaches (matrix inversion)
- Similarity group transformation methods (coupled ordinary differential equations)

All these methods start normally with a Hartree-Fock basis as the calculational basis.

Important ingredients to have in codes

- Be able to validate and verify the algorithms.
- Include concepts like unit testing. Gives the possibility to test and validate several or all parts of the code.
- Validation and verification are then included naturally and one can develop a better attitude to what is meant with an ethically sound scientific approach.

A structured approach to solving problems

In the steps that lead to the development of clean code you should think of:
- How to structure a code in terms of functions (use IDEs or advanced text editors like sublime or atom)
- How to make a module
- How to read input data flexibly from the command line or files
- How to create graphical/web user interfaces
- How to write unit tests
- How to refactor code in terms of classes (instead of functions only)
- How to conduct and automate large-scale numerical experiments
- How to write scientific reports in various formats (LaTeX, HTML, doconce)

Additional benefits

Many of the above aspects will save you a lot of time when you incrementally extend software over time from simpler to more complicated problems. In particular, you will benefit from many good habits:
- New code is added in a modular fashion to a library (modules)
- Programs are run through convenient user interfaces
- It takes one quick command to let all your code undergo heavy testing
- Tedious manual work with running programs is automated,
- Your scientific investigations are reproducible, scientific reports with top quality typesetting are produced both for paper and electronic devices. Use version control software like git and repositories like github

Unit Testing

Unit Testing is the practice of testing the smallest testable parts, called units, of an application individually and independently to determine if they behave exactly as expected. Unit tests (short code fragments) are usually written such that they can be preformed at any time during the development to continually verify the behavior of the code. In this way, possible bugs will be identified early in the development cycle, making the debugging at later stages much easier.

Unit Testing, benefits

There are many benefits associated with Unit Testing, such as:
- It increases confidence in changing and maintaining code. Big changes can be made to the code quickly, since the tests will ensure that everything still is working properly.
- Since the code needs to be modular to make Unit Testing possible, the code will be easier to reuse. This improves the code design.
- Debugging is easier, since when a test fails, only the latest changes need to be debugged.
- Different parts of a project can be tested without the need to wait for the other parts to be available.
- A unit test can serve as a documentation on the functionality of a unit of the code.
Simple example of unit test

Look up the guide on how to install unit tests for C++ at course webpage. This is the version with classes.

```cpp
#include <unittest++/UnitTest++.h>

class MyMultiplyClass {
public:
    double multiply(double x, double y) {
        return x * y;
    }

    TEST(MyMath) {
        MyMultiplyClass my;
        CHECK_EQUAL(56, my.multiply(7,8));
    }

int main() {
    return UnitTest::RunAllTests();
}
```

And without classes

```cpp
#include <unittest++/UnitTest++.h>

double multiply(double x, double y) {
    return x * y;
}

TEST(MyMath) {
    CHECK_EQUAL(56, multiply(7,8));
    int main() {
        return UnitTest::RunAllTests();
    }
}
```

For Fortran users, the link at http://sourceforge.net/projects/fortranxunit/ contains a similar software for unit testing. For Python go to https://docs.python.org/2/library/unittest.html.

Unit tests

There are many types of unit test libraries. One which is very popular with C++ programmers is Catch. Catch is header only. All you need to do is drop the file(s) somewhere reachable from your project - either in some central location you can set your header search path to find, or directly into your project tree itself!

This is a particularly good option for other Open-Source projects that want to use Catch for their test suite.

Examples

Computing factorials

```cpp
inline unsigned int Factorial( unsigned int number ) {
    return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

This will compile to a complete executable which responds to command line arguments. If you just run it with no arguments it will execute all test cases (in this case there is just one), report any failures, report a summary of how many tests passed and failed and return the number of failed tests.

Factorial Example

Simple test where we put everything in a single file

```cpp
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main()
#include "catch.hpp"

inline unsigned int Factorial( unsigned int number ) {
    return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

What did we do (1)?

All we did was

```cpp
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main()
#include "catch.hpp"

inline unsigned int Factorial( unsigned int number ) {
    return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

You can also provide your own implementation of main and drive Catch yourself.
What did we do (2)?

We introduce test cases with the \texttt{TEST\_CASE} macro.
The test name must be unique. You can run sets of tests by specifying a wildcarded test name or a tag expression. All we did was define one identifier and include one header and we got everything.
We write our individual test assertions using the \texttt{REQUIRE} macro.

Unit test summary and testing approach

Three levels of tests
- Microscopic level: testing small parts of code, use often unit test libraries
- Mesoscopic level: testing the integration of various parts of your code
- Macroscopic level: testing that the final result is ok

Summary and recommendations

Some simple hints and tips in order to write clean and clear code
- Spell out the algorithm and have a top-down approach to the flow of data
- Start with coding as close as possible to eventual mathematical expressions
- Use meaningful names for variables
- Split tasks in simple functions and modules/classes
- Functions should return as few as possible variables
- Use unit tests and make sure your codes are producing the correct results
- Where possible use symbolic coding to autogenerate code and check results
- Make a proper timing of your algorithms
- Use version control and make your science reproducible
- Use IDEs or smart editors with debugging and analysis tools.
- Automatize your computations interfacing high-level and compiled languages like C++ and Fortran.

Coding Recommendations

Writing clean and clear code is an art and reflects your understanding of
- derivation, verification, and implementation of algorithms
- what can go wrong with algorithms
- overview of important, known algorithms
- how algorithms are used to solve mathematical problems
- reproducible science and ethics
- algorithmic thinking for gaining deeper insights about scientific problems
Computing is understanding and your understanding is reflected in your abilities to write clear and clean code.

Building a many-body basis

An important step in an FCI code is to construct the many-body basis.
While the formalism is independent of the choice of basis, the effectiveness of a calculation will certainly be basis dependent. Furthermore there are common conventions useful to know.
First, the single-particle basis has angular momentum as a good quantum number. You can imagine the single-particle wavefunctions being generated by a one-body Hamiltonian, for example a harmonic oscillator. Modifications include harmonic oscillator plus spin-orbit splitting, or self-consistent mean-field potentials, or the Woods-Saxon potential which mocks up the self-consistent mean-field. For nuclei, the harmonic oscillator, modified by spin-orbit splitting, provides a useful language for describing single-particle states.
Building a many-body basis

Each single-particle state is labeled by the following quantum numbers:
- Orbital angular momentum \( l \)
- Intrinsic spin \( s = 1/2 \) for protons and neutrons
- Angular momentum \( j = l \pm 1/2 \)
- \( z \)-component \( m \) (or \( m \))
- Some labeling of the radial wavefunction, typically \( n \) the number of nodes in the radial wavefunction, but in the case of harmonic oscillator one can also use the principal quantum number \( N \), where the harmonic oscillator energy is
\[
\langle N = \frac{3}{2} \rangle \hbar \omega.
\]
In this format one labels states by \( n(l) \), with \( l \) replaced by a letter: \( s \) for \( l = 0 \), \( p \) for \( l = 1 \), \( d \) for \( l = 2 \), \( f \) for \( l = 3 \), and thenceforth alphabetical.

Building a many-body basis

There are different kinds of truncations.
- For example, one can start with ‘filled’ orbits (almost always the lowest), and then allow one, two, three... particles excited out of those filled orbits. These are called \( 1p, 2p, 3p \) or \( 1h, 2h, 3h \) excitations.
- Alternatively, one can state a maximal orbit and allow all possible configurations with particles occupying states up to that maximum. This is called full configuration.
- Finally, for particular use in nuclear physics, there is the energy truncation, also called the NilO or \( N_{\text{max}} \) truncation.

Building a many-body basis

In practice the single-particle space has to be severely truncated. This truncation is typically based on the single-particle energies, which is the effective energy from a mean-field potential. Sometimes we freeze the core and only consider a valence space. For example, one may assume a frozen \(^4\)He core, with two protons and two neutrons in the \( 0p_{3/2} \) shell, and then only allow active particles in the \( 0p_{1/2} \) and \( 0p_{3/2} \) orbits.

Another example is a frozen \(^{16}\)O core, with eight protons and eight neutrons filling the \( 0p_{3/2}, 0p_{1/2}, 0d_{3/2}, 0d_{5/2} \) orbits, with valence particles in the \( 0d_{5/2}, 1s_{1/2}, 0d_{3/2} \) orbits.

Sometimes we refer to nuclei by the valence space where their last nucleons go. So, for example, we call \(^{12}\)C a \( 2p \)-shell nucleus, while \(^{28}\)Al is an \( 3d \)-shell nucleus and \(^{56}\)Fe is a \( 2p \)-shell nucleus.

Building a many-body basis

The Hamiltonian matrix will have smaller dimensions (a factor of 10 or more) in the \( J \)-scheme than in the \( M \)-scheme. On the other hand, as we’ll show in the next slide, the \( M \)-scheme is very easy to construct with Slater determinants, while the \( J \)-scheme basis states, and thus the matrix elements, are more complicated, almost always being linear combinations of \( M \)-scheme states. \( J \)-scheme bases are important and useful, but we’ll focus on the simpler \( M \)-scheme.

The quantum number \( m \) is additive (because the underlying group is Abelian): if a Slater determinant \( |\uparrow \downarrow \rangle \) is built from single-particle states all with good \( m \), then the total
\[
M = m_1 + m_2 + m_3 + \ldots
\]
This is not true of \( J \), because the angular momentum group \( SU(2) \) is not Abelian.

Building a many-body basis

In almost all cases, the many-body Hamiltonian is rotationally invariant, meaning it does not depend upon the orientation of the intrinsic motion and the center-of-mass motion factor. In other words, we can know exactly the center-of-mass wavefunction.

For example, one can start with ‘filled’ orbits (almost always \( 1s \)-type). The Hamiltonian will have smaller dimensions and thus the matrix elements, are more complicated, almost always being linear combinations of \( M \)-scheme states.

Another example is a frozen \(^{16}\)O core, with eight protons and eight neutrons filling the \( 0p_{3/2}, 0p_{1/2}, 0d_{3/2} \) orbits, with valence particles in the \( 0d_{5/2}, 1s_{1/2} \) orbits.

Sometimes we refer to nuclei by the valence space where their last nucleons go. So, for example, we call \(^{12}\)C a \( 2p \)-shell nucleus, while \(^{28}\)Al is an \( 3d \)-shell nucleus and \(^{56}\)Fe is a \( 2p \)-shell nucleus.

Building a many-body basis

The upshot is that
- It is easy to construct a Slater determinant with good total \( M \).
- It is trivial to calculate \( M \) for each Slater determinant;
- So it is easy to construct an \( M \)-scheme basis with fixed total \( M \).

Note that the individual \( M \)-scheme basis states will not, in general, have good total \( J \). Because the Hamiltonian is rotationally invariant, however, the eigenstates will have good \( J \). (The situation is muddied when one has states of different \( J \) that are nonetheless degenerate.)
Building a many-body basis

Example: two \( j = 1/2 \) orbits

<table>
<thead>
<tr>
<th>Index</th>
<th>( n )</th>
<th>( l )</th>
<th>( j )</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>-1/2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>-1/2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
</tbody>
</table>

Note that the order is arbitrary.

Building a many-body basis

There are \( \binom{4}{2} = 6 \) two-particle states, which we list with the total \( M \):

<table>
<thead>
<tr>
<th>Occupied</th>
<th>( M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>0</td>
</tr>
<tr>
<td>1,3</td>
<td>-1</td>
</tr>
<tr>
<td>1,4</td>
<td>0</td>
</tr>
<tr>
<td>2,3</td>
<td>0</td>
</tr>
<tr>
<td>2,4</td>
<td>1</td>
</tr>
<tr>
<td>3,4</td>
<td>0</td>
</tr>
</tbody>
</table>

There are 4 states with \( M = 0 \), and 1 each with \( M = \pm 1 \).

Building a many-body basis

As another example, consider using only single particle states from the \( 0d_{5/2} \) space. They have the following quantum numbers

<table>
<thead>
<tr>
<th>Index</th>
<th>( n )</th>
<th>( l )</th>
<th>( j )</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>-5/2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>-3/2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>-1/2</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>1/2</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>3/2</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>5/2</td>
<td>5/2</td>
</tr>
</tbody>
</table>

There are \( \binom{6}{2} = 15 \) two-particle states, which we list with the total \( M \):

<table>
<thead>
<tr>
<th>Occupied</th>
<th>( M )</th>
<th>Occupied</th>
<th>( M )</th>
<th>Occupied</th>
<th>( M )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>-4</td>
<td>2,3</td>
<td>-2</td>
<td>3,5</td>
<td>1</td>
</tr>
<tr>
<td>1,3</td>
<td>-3</td>
<td>2,4</td>
<td>-1</td>
<td>3,6</td>
<td>2</td>
</tr>
<tr>
<td>1,4</td>
<td>-2</td>
<td>2,5</td>
<td>0</td>
<td>4,5</td>
<td>2</td>
</tr>
<tr>
<td>1,5</td>
<td>-1</td>
<td>2,6</td>
<td>1</td>
<td>4,6</td>
<td>3</td>
</tr>
<tr>
<td>1,6</td>
<td>0</td>
<td>3,4</td>
<td>0</td>
<td>5,6</td>
<td>4</td>
</tr>
</tbody>
</table>

There are 3 states with \( M = 0 \), 2 with \( M = 1 \), and so on.

Shell-model Project

The basic goal of this project is for you to build your own configuration-interaction shell-model code. The code will be fairly basic; it will assume that we have a single species of particles, e.g. only neutrons, and you could, if you wish, read in uncoupled two-body matrix elements. Furthermore the pieces of the code will not be the most efficient. Nonetheless it will be usable; most importantly, you will gain a good idea of what goes into a many-body shell-model code.

Shell-model project

The first step is to construct the \( M \)-scheme basis of Slater determinants. Here \( M \)-scheme means the total \( J_z \) of the many-body states is fixed. The steps could be:

- Read in a user-supplied file of single-particle states (examples can be given) or just code these internally;
- Ask for the total \( M \) of the system and the number of particles \( N \);
- Construct all the \( N \)-particle states with given \( M \). You will validate the code by comparing both the number of states and specific states.
The format of a possible input file could be:

<table>
<thead>
<tr>
<th>Index</th>
<th>n</th>
<th>l</th>
<th>2j</th>
<th>2m_j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-5</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-3</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

This represents the $1s^2(1d^2)/2$ valence space, or just the sd space. There are twelve single-particle states, labeled by an overall index, and which have associated quantum numbers the number of radial nodes, the orbital angular momentum $l$, and the angular momentum $j$ and third component $m_j$. To keep everything as integers, we could store $2 \times j$ and $2 \times m_j$.

The next step is to read in the number of particles $N$ and the fixed total $M$ (or, actually, $2 \times M$). For this project we assume only a single species of particles, say neutrons, although this can be relaxed. Note: Although it is often a good idea to try to write a more general code, given the short time allotted we would suggest you keep your ambition in check, at least in the initial phases of the project. You should probably write an error trap to make sure $N$ and $M$ are congruent; if $N$ is even, then $2 \times M$ should be even, and if $N$ is odd then $2 \times M$ should be odd.

Open the file
Read the number of single-particle states (in the above example, 12); allocate memory; all you need is a single array storing $2 \times j$, for each state, labeled by the index.
Read in the quantum numbers and store $2 \times m_j$ (and anything else you happen to want).

The final step is to generate the set of $N$-particle Slater determinants with fixed $M$. The Slater determinants will be stored in occupation representation. Although in many codes this representation is done compactly in bit notation with ones and zeros, but for greater transparency and simplicity we will list the occupied single-particle states. Hence we can store the Slater determinant basis states as $sd(i,j)$, that is an array of dimension $N_{SD}$, the number of Slater determinants, by $N$, the number of occupied state. So if for the $7$th Slater determinant the $2$nd, $3$rd, and $9$th single-particle states are occupied, then $sd(7,1) = 2$, $sd(7,2) = 3$, and $sd(7,3) = 9$.

We can construct an occupation representation of Slater determinants by the odometer method. Consider $N_{SD} = 12$ and $N = 4$. Start with the first $4$ states occupied, that is:
- $sd(1,:) = 1, 2, 3, 4$ (also written as $[1, 2, 3, 4]$)

Now increase the last occupancy recursively:
- $sd(2,:) = 1, 2, 3, 5$
- $sd(3,:) = 1, 2, 3, 6$
- $sd(4,:) = 1, 2, 3, 7$
- $sd(5,:) = 1, 2, 3, 12$
- Then start over with $sd(6,:) = 1, 2, 4, 5$
- and again increase the rightmost digit $sd(7,:) = 1, 2, 4, 6$
- $sd(8,:) = 1, 2, 4, 7$

When we restrict ourselves to an $M$-scheme basis, we could choose two paths. The first is simplest (and simplest is often best, at least in the first draft of a code): generate all possible Slater determinants, and then extract from this initial list a list of those Slater determinants with a given $M$. (You will need to write a short function or routine that computes $M$ for any given occupation.) Alternately, and not too difficult, is to run the odometer routine twice: each time, as a Slater determinant is calculated, compute $M$, but do not store the Slater determinants except the current one. You can then count up the number of Slater determinants with a chosen $M$. Then allocate storage for the Slater determinants, and run the odometer algorithm again, this time storing Slater determinants with the desired $M$ (this can be done with a simple logical flag).
In the shell-model context we can interpret this as 4 $4s_{1/2}$ levels, with $m = \pm 1/2$, we can also think of these are simple four pairs, $\pm k, k = 1, 2, 3, 4$. Later on we will assign single-particle energies, depending on the radial quantum number $n$, that is, $\epsilon_k = |k|\delta$ so that they are equally spaced.

### Space containing six single-particle states

<table>
<thead>
<tr>
<th>Index</th>
<th>n</th>
<th>l</th>
<th>j</th>
<th>m</th>
<th>m_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>

For two particles, there are a total of 15 states, which we list here with the total $M$:

- $|1,2\rangle, M = -4, |1,3\rangle, M = -3$
- $|1,4\rangle, M = -2, |1,5\rangle, M = -1$
- $|1,6\rangle, M = 0, |2,3\rangle, M = -2$
- $|2,4\rangle, M = -1, |2,5\rangle, M = 0$
- $|2,6\rangle, M = 1, |3,4\rangle, M = 0$
- $|3,5\rangle, M = 1, |3,6\rangle, M = 2$
- $|4,5\rangle, M = 2, |4,6\rangle, M = 3$

For $N = 2$ there are 14 states with $M = 0$; show this by hand and confirm your code reproduces it.

### The $(1/2)^4$ space with four single-particle states

<table>
<thead>
<tr>
<th>Index</th>
<th>n</th>
<th>l</th>
<th>j</th>
<th>m</th>
<th>m_s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>

For $N = 2$ there are 4 states with $M = 0$; show this by hand and confirm your code reproduces it.
Example case: pairing Hamiltonian

One can show (and this is part of the project) that

\[
\hat{P} + \hat{P}^\dagger = \sum_{k=1}^{N} \left( \hat{a}^\dagger_k \hat{a}_k + \hat{a}_k \hat{a}^\dagger_k - 1 \right) = \hat{N} - \Omega.
\]

Now define

\[
\hat{P}_z = \frac{1}{2}(\hat{N} - \Omega).
\]

Finally you can show

\[
[\hat{P}_+, \hat{P}_z] = \pm \hat{P}_z.
\]

This means the operators \( \hat{P}_+ \) and \( \hat{P}_z \) form a so-called SU(2) algebra, and we can use all our insights about angular momentum, even though there is no actual angular momentum involved.

So we rewrite the Hamiltonian to make this explicit:

\[
\hat{H} = -G \hat{P}_+ \hat{P}_- - G (\hat{P}^2 - \hat{P}_z^2 + \hat{P}_z^2).
\]

Because of the SU(2) algebra, we know that the eigenvalues of \( \hat{P}_z \) must be of the form \( p(p+1) \), with \( p \) either integer or half-integer, and the eigenvalues of \( \hat{P}_z \) are \( m_z \), with \( m_z \leq |m_z| \), with \( m_z \) also integer or half-integer.

But because \( \hat{P}_z = (1/2)(\hat{N} - \Omega) \), we know that for \( N \) particles the value \( m_z = (N - \Omega)/2 \). Furthermore, the values of \( m_z \) range from \( -\Omega/2 \) (for \( N = 0 \) ) to \( +\Omega/2 \) (for \( N = 2\Omega \), with all states filled)

We deduce the maximal \( p = \Omega/2 \) and for a given \( n \) the values range of \( p \) range from \( (N - \Omega)/2 \) to \( \Omega/2 \) in steps of 1 (for an even number of particles).

Following Racah we introduce the notation \( p = (\Omega - \nu)/2 \) where \( \nu = 0, 1, 3, 4, \ldots, \Omega - N - \Omega \). With this it is easy to deduce that the eigenvalues of the pairing Hamiltonian are

\[
-G(N - \nu(2\Omega + 2 - N - \nu))/4
\]

This also works for \( N \) odd, with \( \nu = 1, 3, 5, \ldots, \Omega - 1 \).
Example case: pairing Hamiltonian

Let’s take a specific example: \( \Omega = 3 \) so there are 6 single-particle states, and \( N = 3 \), with \( v = 1, 3 \). Therefore there are two distinct eigenvalues, \( E = -2G, 0 \)

Now let’s work this out explicitly. The single particle degrees of freedom are defined as

<table>
<thead>
<tr>
<th>States</th>
<th>( n )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-1/2</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>1/2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>-1/2</td>
</tr>
<tr>
<td>4</td>
<td>-2</td>
<td>1/2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>-1/2</td>
</tr>
<tr>
<td>6</td>
<td>-3</td>
<td>1/2</td>
</tr>
</tbody>
</table>

There are \( \binom{6}{3} = 20 \) three-particle states, but there are 9 states with \( M = +1/2 \), namely

\[ |1,2,3\rangle, |1,2,5\rangle, |1,4,6\rangle, |2,3,4\rangle, |2,3,6\rangle, |2,4,5\rangle, |2,5,6\rangle, |3,4,6\rangle, |4,5,6\rangle. \]

Example case: pairing Hamiltonian

In this basis, the operator

\[ \hat{P}_+ = \hat{a}^\dagger_1 \hat{a}^\dagger_2 \hat{a}^\dagger_3 + \hat{a}^\dagger_4 \hat{a}^\dagger_5 \hat{a}^\dagger_6 \]

From this we can determine that

\[ \hat{P}_- |1,4,6\rangle = \hat{P}_- |2,3,6\rangle - \hat{P}_- |2,4,5\rangle = 0 \]

so those states all have eigenvalue 0.

Example case: pairing Hamiltonian

Now for further example,

\[ \hat{P}_- |1,2,3\rangle = |3\rangle \]

so

\[ \hat{P}_+ \hat{P}_- |1,2,3\rangle = |1,2,3\rangle + |3,4,6\rangle \]

without picking up a phase.

Example case: pairing Hamiltonian

Now for further example,

\[ \hat{P}_- |1,2,3\rangle = |3\rangle \]

so

\[ \hat{P}_+ \hat{P}_- |1,2,3\rangle = |1,2,3\rangle + |3,4,6\rangle + |3,5,6\rangle \]

The second term vanishes because state 3 is occupied twice, and reordering the last term we get

\[ \hat{P}_+ \hat{P}_- |1,2,3\rangle = |1,2,3\rangle + |3,5,6\rangle \]

Example case: pairing Hamiltonian

Now we take the following Hamiltonian

\[ \hat{H} = \sum_n \hat{N}_n \hat{P} - G \hat{P} \hat{P}^\dagger \hat{P} \]

where

\[ \hat{N}_n = \hat{N}_n^{\text{def}}_n + 1/2 \hat{N}_n^{\text{def}}_n + 1/2 \]

and

\[ \hat{P}^\dagger = \sum \hat{a}^\dagger_{\alpha, \mu=1/2} \hat{a}_{\alpha, \mu=1/2} \]

We can write down the 6 \( \times 6 \) Hamiltonian in the basis from the prior slide:

\[
\begin{pmatrix}
2G & -G & -G & -G & -G & 0 \\
-G & 4G & -G & -G & -G & -G \\
-G & -G & -G & -G & -G & 0
\end{pmatrix}
\]
Building a Hamiltonian matrix

The goal is to compute the matrix elements of the Hamiltonian, specifically matrix elements between many-body states (Slater determinants) of two-body operators:

$$\sum_{pqrs} V_{pqrs} \hat{a}_p^{\dagger} \hat{a}_q^{\dagger} \hat{a}_s \hat{a}_r$$

In particular we will need to compute

$$\langle \beta | \hat{a}_p^{\dagger} \hat{a}_q^{\dagger} \hat{a}_s \hat{a}_r | \alpha \rangle$$

where $\alpha, \beta$ are indices labeling Slater determinants and $p, q, r, s$ label single-particle states.

Building a Hamiltonian matrix, first step

The first step can take as input an initial Slater determinant (whose position in the list of basis Slater determinants is $i$) written as an ordered list of occupied single-particle states, e.g. $1, 2, 5, 8$, and the indices $p, q, r, s$ from the two-body operator. It will return another final Slater determinant if the single-particle states $r$ and $s$ are occupied, else it will return an empty Slater determinant (all zeroes). If $r$ and $s$ are in the list of occupied single particle states, then replace the initial single-particle states $j$ as $i \rightarrow r$ and $j \rightarrow r$.

Building a Hamiltonian matrix, second step

The second step will take the final Slater determinant from the first step (if not empty), and then order by pairwise permutations (i.e., if the Slater determinant is $i_1, i_2, i_3, \ldots$, then if $i_k > i_{k+1}$, interchange $i_k \leftrightarrow i_{k+1}$.

Note: there are other, more efficient ways to do this than the method we describe, but you will be able to produce a working code quickly.

Building a Hamiltonian matrix

As we coded in the first step, a Slater determinant $|\alpha\rangle$ with index $\alpha$ is a list of $N$ occupied single-particle states $i_1 < i_2 < i_3 \ldots < i_N$. Furthermore, for the two-body matrix elements $V_{pqrs}$ we normally assume $p < q$ and $r < s$. For our specific project, the interaction is much simpler and you can use this to simply considerably the setup of a shell-model code for project 2.

What follows here is a more general, but still brute force, approach.

Building a Hamiltonian matrix

Write a function that:
- Has as input the single-particle indices $p, q, r, s$ for the two-body operator and the index $\alpha$ for the ket Slater determinant;
- Returns the index $\beta$ of the unique (if any) Slater determinant such that
  $$|\beta\rangle = \hat{a}_p^{\dagger} \hat{a}_q^{\dagger} \hat{a}_s \hat{a}_r |\alpha\rangle$$
  as well as the phase
- This is equivalent to computing
  $$\langle \beta | \hat{a}_p^{\dagger} \hat{a}_q^{\dagger} \hat{a}_s \hat{a}_r |\alpha\rangle$$

Building a Hamiltonian matrix

It will also output a phase. If any two single-particle occupancies are repeated, the phase is 0. Otherwise it is +1 for an even permutation and -1 for an odd permutation to bring the final Slater determinant into ascending order, $j_1 < j_2 < j_3 \ldots$. 

Building a Hamiltonian matrix

In particular we will need to compute

$$\langle \beta | \hat{a}_p^{\dagger} \hat{a}_q^{\dagger} \hat{a}_s \hat{a}_r |\alpha\rangle$$

where $\alpha, \beta$ are indices labeling Slater determinants and $p, q, r, s$ label single-particle states.
You will also need to include the single-particle energies. This is easy: they only contribute to diagonal matrix elements, that is, \( H(\alpha, \alpha) \). Simply find the occupied single-particle states \( i \) and add the corresponding \( \epsilon(i) \).

Consider the many-body state \( \Psi_\lambda \) expressed as linear combinations of Slater determinants (SD) of orthonormal single-particle states \( \varphi(i) \):

\[
\Psi_\lambda = \sum_j C_{\lambda j} SD_j 
\]

(2)

Using the Slater-Condon rules the matrix elements of any one-body \( (\hat{O}_1) \) or two-body \( (\hat{O}_2) \) operator expressed in the determinant space have simple expressions involving one- and two-fermion integrals in our given single-particle basis. The diagonal elements are given by:

\[
\langle SD|\hat{O}_1|SD\rangle = \sum_{i\in SD} \langle \varphi(i)|\hat{O}_1|\varphi(i) \rangle
\]

(3)

\[
\langle SD|\hat{O}_2|SD\rangle = \frac{1}{2} \sum_{(i,j)\in SD} \{ \langle \varphi(i)|\hat{O}_2|\varphi(j) \rangle - \langle \varphi(j)|\hat{O}_2|\varphi(i) \rangle \}
\]

(4)
Hamiltonian matrix without the bit representation, one and two-body operators

For two determinants which differ only by the substitution of single-particle states \(i\) with a single-particle state \(j\):

\[
\langle SD(O_i|SD_j) = \langle \phi_i|O_i|\phi_j \rangle (4)
\]

For two determinants which differ by two single-particle states:

\[
\begin{align*}
\langle SD(O_i|SD_j) &= 0 \\
\langle SD(O_i|SD_j) &= \langle \phi_{i1}\phi_{i2}|O_1O_2|\phi_{j1}\phi_{j2} \rangle 
\end{align*}
\]

All other matrix elements involving determinants with more than two substitutions are zero.

Computing expectation values and transitions in the shell-model

When we diagonalize the Hamiltonian matrix, the eigenvectors are the coefficients \(C_{\lambda j}\) used to express the many-body state \(\Psi_\lambda\) in terms of a linear combination of Slater determinants (SD) of orthonormal single-particle states \(\phi_r\).

With these eigenvectors we can compute the transition likelihood of a one-body operator as:

\[
\langle \Psi_\lambda|C_{\lambda j}|\Psi_{\sigma} \rangle = \sum_{0} C_{\lambda j} C_{\lambda j} \langle SD_i|a^\dagger_{pq}|SD_j \rangle.
\]

Writing the one-body operator in second quantization as:

\[
C_{\lambda 1} = \sum_{pq} \langle \phi_i|\phi_q \rangle a_{i} a_{q},
\]

we have:

\[
\langle \Psi_\lambda|C_{\lambda 1}|\Psi_{\sigma} \rangle = \sum_{pq} \langle \phi_i|\phi_q \rangle \sum_{0} C_{\lambda j} C_{\lambda j} \langle SD_i|a^\dagger_{pq}|SD_j \rangle.
\]

Strategies for setting up an algorithm

An efficient implementation of these rules requires:

- to find the number of single-particle state substitutions between two determinants
- to find which single-particle states are involved in the substitution
- to compute the phase factor if a reordering of the single-particle states has occurred

We can solve this problem using our odometric approach or alternatively using a bit representation as discussed below and in more detail in:

- Scemama and Gimer’s article (Fortran codes)
- Simen Kvaal’s article on how to build an FCI code (C++ code)

We recommend in particular the article by Simen Kvaal. It contains nice general classes for creation and annihilation operators as well as the calculation of the phase (see below).

Operators in second quantization

In the buildup of a shell-model or FCI code that is meant to tackle large dimensionalities we need to deal with the action of the Hamiltonian \(H\) on a Slater determinant represented in second quantization as:

\[
|\alpha_1\ldots\alpha_n\rangle = a_{1}^\dagger\ldots a_{n}^\dagger |0\rangle.
\]

The time consuming part stems from the action of the Hamiltonian on the above determinant,

\[
\left(\sum_{i<j} |t_{ij}|^2 a_{i}^\dagger a_{j} + \sum_{i>j} |t_{ij}|^2 a_{i} a_{j}^\dagger \right) a_{1}^\dagger\ldots a_{n}^\dagger |0\rangle.
\]

A practically useful way to implement this action is to encode a Slater determinant as a bit pattern.

Computing expectation values and transitions in the shell-model and spectroscopic factors

The terms we need to evaluate then are just the elements:

\[
\langle SD_i|a^\dagger_{pq}|SD_j \rangle
\]

which can be rewritten in terms of spectroscopic factors by inserting a complete set of Slater determinants as:

\[
\langle SD_i|a^\dagger_{pq}|SD_j \rangle = \sum_{\lambda} \langle SD_i|\lambda\rangle a^\dagger_{\lambda p} \langle \lambda|SD_j \rangle,
\]

where \(\langle SD_i|\lambda\rangle a^\dagger_{\lambda p} \langle \lambda|SD_j \rangle\) are the spectroscopic factors. These can be easily evaluated in m-scheme. Using the Wigner-Eckart theorem we can transform these to a \(J\)-coupled scheme through so-called reduced matrix elements.

Operators in second quantization

Assume that we have at our disposal \(n\) different single-particle states \(\alpha_0,\alpha_1,\ldots,\alpha_{n-1}\) and that we can distribute among these states \(N\leq n\) particles.

A Slater determinant can then be coded as an integer of \(n\) bits. As an example, if we have \(n=16\) single-particle states \(\alpha_0,\alpha_1,\ldots,\alpha_{15}\) and \(N=4\) fermions occupying the states \(\alpha_9,\alpha_8,\alpha_{12}\) and \(\alpha_{13}\) we could write this Slater determinant as:

\[
\Phi = a_{9}^\dagger a_{8}^\dagger a_{12}^\dagger a_{13}^\dagger |0\rangle.
\]

The unoccupied single-particle states have bit value 0 while the occupied ones are represented by bit state 1. In the binary notation we would write this 16 bits long integer as:

\[
0001010001001000000111010000010001000011000010001111000001111000
\]

which translates into the decimal number:

\[
23 + 26 + 210 + 213 = 9288.
\]

We can thus encode a Slater determinant as a bit pattern.
Operators in second quantization

We have an SD representation

\[ \Phi_0 = a_{\alpha_0}^\dagger a_{\alpha_1}^\dagger a_{\alpha_2}^\dagger a_{\alpha_3}^\dagger |0\rangle, \]

in a more compact way as

\[ \Phi_{0,3,10,13} = |0010010001000100\rangle. \]

The total number of bit patterns is \(2^n\).

Operators in second quantization

Consider the action of \(a_{\alpha_3}\) on various Slater determinants:

\[ a_{\alpha_3} \Phi_{0111} = a_{\alpha_3}^\dagger |0111\rangle = 0 \times |0111\rangle, \]
\[ a_{\alpha_3} \Phi_{0101} = a_{\alpha_3}^\dagger |0101\rangle = (-1) \times |0101\rangle, \]
\[ a_{\alpha_3} \Phi_{0100} = a_{\alpha_3}^\dagger |0100\rangle = 0 \times |0100\rangle, \]
\[ a_{\alpha_3} \Phi_{1001} = a_{\alpha_3}^\dagger |1001\rangle = (-1) \times |1001\rangle, \]
\[ a_{\alpha_3} \Phi_{1010} = a_{\alpha_3}^\dagger |1010\rangle = 0 \times |1010\rangle, \]
\[ a_{\alpha_3} \Phi_{1011} = a_{\alpha_3}^\dagger |1011\rangle = (-1) \times |1011\rangle, \]
\[ a_{\alpha_3} \Phi_{1100} = a_{\alpha_3}^\dagger |1100\rangle = (-1) \times |1100\rangle, \]
\[ a_{\alpha_3} \Phi_{1101} = a_{\alpha_3}^\dagger |1101\rangle = 0 \times |1101\rangle, \]
\[ a_{\alpha_3} \Phi_{1110} = a_{\alpha_3}^\dagger |1110\rangle = (-1) \times |1110\rangle, \]
\[ a_{\alpha_3} \Phi_{1111} = a_{\alpha_3}^\dagger |1111\rangle = 0 \times |1111\rangle. \]

What is the simplest way to obtain the phase when we act with one annihilation (creation) operator on the given Slater determinant representation?

Operators in second quantization

Similarly

\[ a_{\alpha_3}^\dagger \Phi_{0,3,10,13} = a_{\alpha_3}^\dagger |0010010001000100\rangle = a_{\alpha_3}^\dagger a_{\alpha_3}^\dagger a_{\alpha_1}^\dagger a_{\alpha_3}^\dagger |0\rangle, \]

which becomes

\[ -a_{\alpha_3}^\dagger a_{\alpha_3}^\dagger a_{\alpha_1}^\dagger a_{\alpha_3}^\dagger |0\rangle = 0! \]

This gives a simple recipe:

- If one of the bits \(b_j\) is 1 and we act with a creation operator on this bit, we return a null vector
- If \(b_j = 0\), we set it to 1 and return a sign factor \((-1)^j\), where \(j\) is the number of bits set before bit \(j\).

Operators in second quantization

The action

\[ a_{\alpha_3}^\dagger a_{\alpha_4}^\dagger \Phi_{0,3,8,10,13} = a_{\alpha_3}^\dagger a_{\alpha_4}^\dagger a_{\alpha_3}^\dagger a_{\alpha_3}^\dagger a_{\alpha_1}^\dagger a_{\alpha_3}^\dagger |0\rangle, \]

in a more compact way as

\[ \Phi_{0,3,8,10,13} = |0010010001000100\rangle. \]

The action of \(a_{\alpha_3}^\dagger a_{\alpha_4}^\dagger \Phi_{0,3,8,10,13}\) can be obtained by subtracting the logical sum (AND operation) of \(\Phi_{0,3,8,10,13}\) and a word which represents only \(\alpha_4\), that is

\[ |00000000000000\rangle. \]

from \(\Phi_{0,3,8,10,13}\) = |0010010001000100\rangle.

This operation gives |0001001000100000\rangle. Similarly, we can form \(a_{\alpha_3} a_{\alpha_4} \Phi_{0,3,8,10,13}\), say, by adding |0001001000100000\rangle to \(a_{\alpha_3} \Phi_{0,3,8,10,13}\), first checking that their logical sum is zero in order to make sure that the state \(\alpha_4\) is not already occupied.
Operators in second quantization

It is trickier however to get the phase \((-1)^l\). One possibility is as follows:

- Let $S_1$ be a word that represents the 1-bit to be removed and all others set to zero.
- In the previous example $S_1 = \{0000000000000000\}$
- Define $S_2$ as the similar word that represents the bit to be added, that is in our case $S_2 = \{0001000000000000\}$.
- Compute then $S = S_1 - S_2$, which here becomes $S = \{0111000000000000\}$.
- Perform then the logical AND operation of $S$ with the word containing $\Phi_{2.3.8.10.13} = \{0010010001001000\}$, which results in $\{0001000000000000\}$. Counting the number of 1 bits gives the above. Here you need however an algorithm for bitcounting.

Eigenvalue problems, basic definitions

Let us consider the matrix $A$ of dimension $n$. The eigenvalues of $A$ are defined through the matrix equation

\[ Ax^{(\nu)} = \lambda^{(\nu)} x^{(\nu)}, \]

where $\lambda^{(\nu)}$ are the eigenvalues and $x^{(\nu)}$ the corresponding eigenvectors. Unless otherwise stated, when we use the wording eigenvector we mean the right eigenvector. The left eigenvalue problem is defined as

\[ x^{(\nu)} A = \lambda^{(\nu)} x^{(\nu)}. \]

The above right eigenvector problem is equivalent to a set of $n$ equations with $n$ unknowns $x_i$.

Abel-Ruffini Impossibility Theorem

The Abel-Ruffini theorem (also known as Abel’s impossibility theorem) states that there is no general solution in radicals to polynomial equations of degree five or higher.

The content of this theorem is frequently misunderstood. It does not assert that higher-degree polynomial equations are unsolvable. In fact, if the polynomial has real or complex coefficients, and we allow complex solutions, then every polynomial equation has solutions; this is the fundamental theorem of algebra. Although these solutions cannot always be computed exactly with radicals, they can be computed to any desired degree of accuracy using numerical methods such as the Newton-Raphson method or Laguerre method, and in this way they are no different from solutions to polynomial equations of the second, third, or fourth degrees.

The theorem only concerns the form that such a solution must take. The content of the theorem is that the solution of a higher-degree equation cannot in all cases be expressed in terms of the polynomial coefficients with a finite number of operations of addition, subtraction, multiplication, division and root extraction.

Bit counting

We include here a python program which may aid in this direction. It uses bit manipulation functions from http://wiki.python.org/moin/BitManipulation.

```python
import math

def annihilate(self, j):
    # Annihilate the j'th bit.
    return self.word & ~2**j

def create(self, j):
    # Create the j'th bit.
    return self.word | 2**j

def __init__(self):
    # Class for Slater determinants
    self.word = 0

# Bit-manipulation stolen from:
# A simple Python class for Slater determinant manipulation
# Bit-manipulation stolen from:
# Bit counting

def testBit(self, offset):
    # Test if bit offset is set.
    int_type = self.word
    # Test if bit offset is set.
    return int_type & 2**offset

def setBit(int_type, offset):
    # Set the offset'th bit.
    return int_type | 2**offset

def toggleBit(self, offset):
    # Toggle the offset'th bit.
    return int_type ^ 2**offset

def count(self):
    # Count the number of bits set.
    count = 0
    while(int_type):
        count += 1
    return count

def testBit(self, offset):
    # Test if bit offset is set.
    int_type = self.word
    # Test if bit offset is set.
    return int_type & 2**offset

def setBit(int_type, offset):
    # Set the offset'th bit.
    return int_type | 2**offset

def toggleBit(self, offset):
    # Toggle the offset'th bit.
    return int_type ^ 2**offset

def count(self):
    # Count the number of bits set.
    count = 0
    while(int_type):
        count += 1
    return count

s = phi.create(7)
s = phi.annihilate(2)
s = phi.create(200)
```
### Abel-Ruffini Impossibility Theorem

The Abel-Ruffini theorem says that there are some fifth-degree equations whose solution cannot be so expressed. The equation \( x^5 - x + 1 = 0 \) is an example. Some other fifth-degree equations can be solved by radicals, for example \( x^5 - x^4 - x + 1 = 0 \). The precise criterion that distinguishes between those equations that can be solved by radicals and those that cannot was given by Galois and is now part of Galois theory: a polynomial equation can be solved by radicals if and only if its Galois group is a solvable group. Today, in the modern algebraic context, we say that second, third and fourth degree polynomial equations can always be solved by radicals because the symmetric groups \( S_2, S_3 \) and \( S_4 \) are solvable groups, whereas \( S_n \) is not solvable for \( n \geq 5 \).

### Eigenvalue problems, basic definitions

In the present discussion we assume that our matrix is real and symmetric, that is \( A \in \mathbb{R}^{n \times n} \). The matrix \( A \) has \( n \) eigenvalues \( \lambda_1, \ldots, \lambda_n \) (distinct or not). Let \( D \) be the diagonal matrix with the eigenvalues on the diagonal:

\[
D = \begin{pmatrix}
\lambda_1 & 0 & 0 & \cdots & 0 \\
0 & \lambda_2 & 0 & \cdots & 0 \\
0 & 0 & \lambda_3 & \cdots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & \cdots & \lambda_n
\end{pmatrix}
\]

If \( A \) is real and symmetric then there exists a real orthogonal matrix \( S \) such that

\[ S^T A S = \text{diag}(\lambda_1, \ldots, \lambda_n), \]

and for \( j = 1 : n \) we have \( A S(:,j) = \lambda_j S(:,j) \).

### Eigenvalue problems, basic definitions

To obtain the eigenvalues of \( A \in \mathbb{R}^{n \times n} \), the strategy is to perform a series of similarity transformations on the original matrix \( A \), in order to reduce it either into a diagonal form as above or into a tridiagonal form. We say that a matrix \( B \) is a similarity transform of \( A \) if

\[ B = S^T A S, \quad \text{where} \quad S^T S = S^{-1} S = I. \]

The importance of a similarity transformation lies in the fact that the resulting matrix has the same eigenvalues, but the eigenvectors are in general different.

### Eigenvalue problems, basic definitions

The basic philosophy is to

- Either apply subsequent similarity transformations (direct method) so that
  \[ S^T_j A S_{j-1} \cdots S_2 A S_1 = D, \quad (7) \]
- Or apply subsequent similarity transformations so that \( A \) becomes tridiagonal (Householder) or upper/lower triangular (the QR method to be discussed later).
- Thereafter, techniques for obtaining eigenvalues from tridiagonal matrices can be used.
- Or use so-called power methods.
- Or use iterative methods (Krylov, Lanczos, Arnoldi). These methods are popular for huge matrix problems.

### Discussion of methods for eigenvalues

The general overview

One speaks normally of two main approaches to solving the eigenvalue problem.

- The first is the formal method, involving determinants and the characteristic polynomial. This proves how many eigenvalues there are, and is the way most of you learned about how to solve the eigenvalue problem, but for matrices of dimensions greater than 2 or 3, it is rather impractical.
- The other general approach is to use similarity or unitary transformations to reduce a matrix to diagonal form. This is normally done in two steps: first reduce to for example a tridiagonal form, and then to diagonal form. The main algorithms we will discuss in detail, Jacobi’s and Householder’s (so-called direct method) and Lanczos algorithms (an iterative method), follow this methodology.
Eigenvalues methods

Direct or non-iterative methods require for matrices of dimensionality $n \times n$ typically $O(n^3)$ operations. These methods are normally called standard methods and are used for dimensionality $n \leq 10^2$ or smaller. A brief historical overview

<table>
<thead>
<tr>
<th>Year</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950</td>
<td>20</td>
</tr>
<tr>
<td>1965</td>
<td>200</td>
</tr>
<tr>
<td>1980</td>
<td>4000</td>
</tr>
<tr>
<td>1995</td>
<td>5000</td>
</tr>
</tbody>
</table>

shows that in the course of 80 years the dimension that direct diagonalization methods can handle has increased by almost a factor of $10^3$ (note this is for serial versions). However, it pales beside the progress achieved by computer hardware, from flops to petaflops, a factor of almost $10^{12}$. We see clearly played out in history the $O(n^3)$ bottleneck of direct matrix algorithms. Sloppily speaking, when $n \sim 10^4$ is cubed we have $O(10^{12})$ operations, which is smaller than the $10^{15}$ increase in flops.

Discussion of methods for eigenvalues

If the matrix to diagonalize is large and sparse, direct methods simply become impractical, also because many of the direct methods tend to destroy sparsity. As a result large dense matrices may arise during the diagonalization procedure. The idea behind iterative methods is to project the $n \times n$-dimensional problem to smaller spaces, so-called Krylov subspaces. Given a matrix $A$ and a vector $v$, the associated Krylov sequences of vectors (and thereby subspaces) $A v, A^2 v, A^3 v, \ldots$ represent successively larger Krylov subspaces.

<table>
<thead>
<tr>
<th>Method</th>
<th>$A = A^*$</th>
<th>$A \neq A^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conjugate gradient</td>
<td>Laplacian</td>
<td>Arnoldi</td>
</tr>
<tr>
<td>GMRES etc</td>
<td>Laplacian</td>
<td>Arnoldi</td>
</tr>
</tbody>
</table>

Eigenvalues and Lanczos’ method

Basic features with a real symmetric matrix (and normally huge $n > 10^6$ and sparse) $A$ of dimension $n \times n$.

- Lanczos’ algorithm generates a sequence of real tridiagonal matrices $T_k$ of dimension $k \times k$ with $k \leq n$, with the property that the extremal eigenvalues of $T_k$ are progressively better estimates of $A$ extremal eigenvalues.

- The similarity transformation is $T = Q^* A Q$.

- With the first vector $Q_0 = v_1$, we are going to solve iteratively

  $T = Q^* A Q$,

  with the first vector $Q_0 = v_1$. We can write out the matrix $Q$ in terms of its column vectors

  $Q = \begin{bmatrix} v_1 & \cdots & v_k & \cdots & v_n \end{bmatrix}$

  Using the fact that $Q Q^* = I$, we can rewrite

  $T = Q^* A Q$,

  as

  $Q T = A Q$.

Eigenvalues and Lanczos’ method, tridiagonal matrix

The matrix

$T = Q^* A Q$,

can be written as

$T = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \cdots \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \cdots \\ 0 & \cdots & 0 & \beta_{k-2} & \alpha_{k-1} & \beta_{k-1} \\ 0 & \cdots & 0 & 0 & \beta_{k-1} & \alpha_k \end{bmatrix}$

Eigenvalues and Lanczos’ method, tridiagonal and orthogonal matrices

If we equate columns

$T = \begin{bmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \cdots \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \cdots \\ 0 & \cdots & 0 & \beta_{k-2} & \alpha_{k-1} & \beta_{k-1} \\ 0 & \cdots & 0 & 0 & \beta_{k-1} & \alpha_k \end{bmatrix}$

we obtain

$A Q_k = \beta_k Q_{k-1} + \alpha_k Q_k + \beta_k Q_{k+1}$.
Eigenvalues and Lanczos’ method, defining the Lanczos’ vectors

We have thus
\[ \hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k \hat{q}_k + \beta_k \hat{q}_{k+1}, \]
with \( \beta_0 \hat{q}_0 = 0 \) for \( k = 1 : n - 1 \). Remember that the vectors \( \hat{q}_k \) are orthonormal and this implies
\[ \alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k, \]
and these vectors are called Lanczos vectors.

Eigenvalues and Lanczos’ method, basic steps

We have thus
\[ \hat{A}\hat{q}_k = \beta_{k-1}\hat{q}_{k-1} + \alpha_k \hat{q}_k + \beta_k \hat{q}_{k+1}, \]
with \( \beta_0 \hat{q}_0 = 0 \) for \( k = 1 : n - 1 \) and
\[ \alpha_k = \hat{q}_k^T \hat{A}\hat{q}_k. \]
If
\[ \hat{r}_k = (\hat{A} - \alpha_k \hat{I})\hat{q}_k - \beta_{k-1}\hat{q}_{k-1}, \]
is non-zero, then
\[ \hat{q}_{k+1} = \hat{r}_k / \beta_k, \]
with \( \beta_k = \pm \|\hat{r}_k\|_2. \)