

BIGSTICK: A flexible configuration-interaction shell-model code ¹

Calvin W. Johnson², W. Erich Ormand³, Kenneth S. McElvain⁴, and Hongzhang Shan⁵

September 4, 2017

¹UCRL number: xxxxx

²Department of Physics, San Diego State University, 5500 Campanile Drive, San Diego CA 92182-1233

³Lawrence Livermore National Laboratory, P.O. Box 808, L-414, Livermore, CA 94551

⁴Department of Physics, University of California, Berkeley 366 Leconte Hall MC 7300, Berkeley, CA 94720

⁵Computational Research Division, Lawrence Berkeley Laboratory, Berkeley, CA, 94720

Abstract

We present **BIGSTICK**, a flexible configuration-interaction open-source shell-model code for the many-fermion problem. Written mostly in Fortran 90 with some later extensions, **BIGSTICK** utilizes a factorized on-the-fly algorithm for computing many-body matrix elements, and has both MPI (distributed memory) and OpenMP (shared memory) parallelization, and can run on platforms ranging from laptops to the largest parallel supercomputers. It uses a flexible yet efficient many-body truncation scheme, and reads input files in multiple formats, allowing one to tackle both phenomenological (major valence shell space) and *ab initio* (the so-called *no-core shell model*) calculations. **BIGSTICK** can generate energy spectra, transition and static one-body densities, and expectation values of scalar operators. Using the built-in Lanczos algorithm one can compute transition probability distributions and decompose wave functions into components defined by group theory.

This manual provides a general guide to compiling and running **BIGSTICK**, which comes with numerous sample input files, as well as some of the basic theory underlying the code. The code also comes with an incomplete “inside guide” which provides more details into the inner workings.

This code is distributed under an Open Source License GPL.v2. The source code and sample inputs are found at github.com/cwjsdsu/BigstickPublick.

Contents

1	Introduction	4
1.1	Expectations of users	5
1.2	License and how to cite	5
1.3	Reporting bugs and other issues	6
1.4	A brief history of BIGSTICK, and acknowledgements	6
2	How we solve the many-body problem	8
2.1	Matrix formulation of the Schrödinger equation	9
2.2	Representation of the basis	12
2.2.1	Factorization of the basis	14
2.3	The Lanczos algorithm and computational cost	16
2.4	Representation of the Hamiltonian	18
2.5	An incomplete survey of other codes	19
3	Getting started with BIGSTICK	22
3.1	What can BIGSTICK do?	22
3.2	Downloading and compiling the code	23
3.3	Required input files	24
3.4	Running the code	25
3.5	Some sample runs	29
3.6	Typical run times	29
4	Using BIGSTICK, in detail	31
4.1	Overview of input files	31
4.2	Defining the model space	32
4.2.1	Particle-hole conjugation	35
4.2.2	Truncation of the many-body space	36
4.2.3	Advanced truncation options	39
4.2.4	How to handle ‘different’ proton-neutron spaces	40
4.3	Interaction files	40
4.3.1	Scaling and autoscaling	42
4.3.2	Proton-neutron and other isospin-breaking formats	42
4.3.3	MFDn format input	45
4.3.4	Three-body forces	46

4.4	Primary runtime options	47
4.4.1	Autoinput	47
4.4.2	Standard or normal runs	47
4.4.3	One-body density matrices and occupations	47
4.4.4	Other primary options	49
4.5	Diagonalization options	49
4.5.1	Convergence	51
4.6	Secondary runtime options	53
4.6.1	Expectation value	54
4.6.2	Applying a one-body transition operator	54
4.6.3	Applying a two-body body scalar operator	55
4.6.4	Generating strength function distributions	56
4.6.5	Overlap or dot product of wave functions	57
4.7	Output files	57
4.7.1	Secondary files	58
4.7.2	Diagnosis files	59
4.8	Memory usage	59
5	Applications	60
5.1	One-body density matrices	60
5.1.1	Symmetries of density matrix elements	62
5.1.2	Particle occupations from densities	62
5.1.3	Strengths from density matrix elements	62
5.1.4	Sample case: spin-flip	64
5.1.5	Charge-changing transitions	65
5.1.6	Sample case: ^{19}F	66
5.2	Strength function option	66
5.2.1	Decomposition	66
5.2.2	Transition strength function distributions: the basics . . .	69
5.2.3	Transitions with good angular momentum	71
5.2.4	Gamow-Teller with strength function option	73
6	A peek behind the curtain	76
6.1	A normal run	76
6.2	Explicit representation of the basis	77
6.3	Explicit representation of the Hamiltonian	79
7	Lanczos algorithm	80
7.1	Standard Lanczos algorithm	80
7.2	Thick-restart Lanczos	82
7.3	Can I restart standard Lanczos?	84

8	Parallel computing	85
8.1	MPI	86
8.1.1	Fragments	86
8.1.2	Opbundles and optypes	87
8.1.3	Modeling	88
8.2	OpenMP	88
9	Timing	90
9.1	Mode times	90
9.2	Timing for parallel runs	91
A	Reduced matrix elements	92
B	The Hamiltonian and other operators in second quantization	93
B.1	Symmetries of matrix elements	94
C	Highlighted references	96

Chapter 1

Introduction

There are many approaches to the quantum many-body problem. **BIGSTICK** is a configuration-interaction many-fermion code, written in Fortran 90. It solves for low-lying eigenvalues of the Hamiltonian of a many fermion system; it does this by creating a basis of many-body states of Slater determinants (actually, the occupation representation of Slater determinants). The Slater determinants are antisymmetrized products of single-particle states with good angular momentum, typically derived from some shell-model-like potential; hence we call this a shell-model basis. The Hamiltonian is assumed to be rotationally invariant and to conserve parity, and is limited to two- and, optionally, three-body forces. Otherwise no assumptions are made about the form of the single-particle states or of the Hamiltonian.

The capabilities of **BIGSTICK** will be detailed below, but in addition to calculating the energy spectra and occupation-space wavefunctions, it can compute particle occupations, expectation values of operators, and static and transition densities and strengths. Most of the applications to date have been in low-energy nuclear physics, but in principle any many-fermion system with two fixed ‘species’ and rotational symmetry can be addressed by **BIGSTICK**, such as the electronic structure of atoms and cold fermionic gases in a spherically symmetric trap; although we have yet to publish papers, we have carried out demonstration calculations for such systems, with ‘spin-up’ and ‘spin-down’ replacing ‘proton’ and ‘neutron.’ We apologize to any atomic physicist who will have to translate our terminology.

In this next chapter we review the basic many-body problem. Chapter 2 outlines the configuration-interaction method and discusses in broad strokes the principles of the algorithms in **BIGSTICK**. Chapter 3 gives an introduction to how to compile and run **BIGSTICK**, while Chapter 4 goes into running the code more detail. **If you are interested in running BIGSTICK immediately, go directly to Chapter 3.**

In this manual we do not give substantial information on the inner workings of the code. With the distribution you will find an as-yet incomplete *Inside Guide* which provide many details. In addition, the code itself is heavily com-

mented. While internal information in **BIGSTICK** is highly compressed through factorization, a technique outlined in Chapter 2, it is possible to get out explicit representations of the many-body basis states and the many-body Hamiltonian matrix; see Chapter 6. Chapter 7 discusses our use of the Lanczos algorithm.

Finally, parallel capabilities of the code is discussed in Chapter 8.

1.1 Expectations of users

Who do we expect to use **BIGSTICK**, and how do we expect them to use it? We designed **BIGSTICK** to be run on a variety of platforms, from laptops to leadership-class supercomputers. We also imagined, and tried to design, **BIGSTICK** for a spectrum of users, with various expectations of them.

A crucial point for any and all users: **BIGSTICK requires at least two kinds of input files to run**, a description of the single-particle space and a file of interaction matrix elements. While we supply with the distribution a number of example input files, it is important for both novice and routine users to understand that such examples are just the **beginning** and not the sum of nuclear physics. **In general it is up to the user to provide interaction files.** We can use the `.int` interaction files usable by NuShell/NuShellX as well as the interaction files used by MFDn. We also provide (?? Ken?) a code for translating files formatted for ANTOINE into files **BIGSTICK** can read.

It is also equally important to not ask **BIGSTICK** to be smarter than you are. While **BIGSTICK** employs many error traps to avoid or at least flag the most common mistakes, the principle of “garbage in, garbage out” still applies: **BIGSTICK** will run correctly given its input, but if the input is flawed, **BIGSTICK** cannot correct for that. This advice applies of course to all computational tools.

While this manual provides a fairly comprehensive introduction to running **BIGSTICK**, it is *not* a detailed tutorial in configuration-interaction methods, the atomic or nuclear shell models, or to basic nuclear physics. We therefore expect the reader to, above all, be comfortable with non-relativistic quantum mechanics (i.e., to fully understand the Schrödinger equation and with Dirac’s bra-ket notation), and to be fluent of the ideas and terminology of the shell model, especially the nuclear shell model, and to understand the basic principles of configuration-interaction methods. We review the latter in the opening of Chapter 2, so that is a good place to start to check your level of comfort. We do suggest references in Appendix C if you need to bring your knowledge up to speed.

1.2 License and how to cite

This code is distributed under an Open Source License GPL.v2. For a full description of this license, see <https://opensource.org/licenses/gpl-2.0.php>. We do not warranty this code or its results. You may copy and distribute verbatim the source code, as long as it is under the same license. You may modify

the code. If you distribute a modified version, you must include a prominent disclaimer noting the date and scope of modifications. (If you make significant and broadly useful changes to the code such as better efficiencies or new capabilities, we'd love to hear about it and possibly incorporate them.)

If you successfully use **BIGSTICK** in your research, please use the following citations:

- C. W. Johnson, W. E. Ormand, and P. G. Krastev, *Comp. Phys. Comm.* **184**, 2761-2774 (2013). (You can also find this article at arXiv:1303.0905.)
- C. W. Johnson, W. E. Ormand, K. S. McElvain, and H. Z. Shan, UCRL-TR-XXXXXX, arXiv:XXXXX (this report)

The first paper, Johnson et al. [2013], in particular discusses the underlying factorized on-the-fly algorithm. This document focuses instead on how to run **BIGSTICK**.

1.3 Reporting bugs and other issues

If you run into trouble, *first read this manual*. Most issues are caused by mistakes in setting up input files, in particular inconsistencies between the single-particle space defined and the interaction file(s). Second, please *read the output carefully*: we have striven to write detailed error traps and often **BIGSTICK** will notify the user of problems. Try running the sample cases and make sure they run to correct completion and that you understand the inputs.

If, having exhausted all the resources presented here, you still have a problem, you may send your issue to Calvin Johnson, cjohnson@mail.sdsu.edu. In particular send a copy of the *entire* output written to screen, which often contains important clues, the input files, and all output files with the extensions **.res**, **.log** and **.bigstick**. Although we hope to be able to help, we cannot guarantee it.

As discussed elsewhere, **BIGSTICK** is developed for Linux and Linux-like environments such as Mac OS X. Although it has a user-friendly menu-driven interface, it still assumes a reasonable facility with many-body physics and in particular low-energy nuclear physics.

Development of **BIGSTICK** is ongoing. We hope to release future versions of the code as additional major capabilities come on line.

1.4 A brief history of **BIGSTICK**, and acknowledgements

In 1997, when two of us (Ormand and Johnson) were both at Louisiana State University, we decided to write our own many-fermion configuration-interaction code christened **REDSTICK**, English for *Baton Rouge*. Over the next decade **REDSTICK** evolved and improved. Most important were the addition of three-body forces and parallelization. As it approached the ten-year mark, we noticed certain limitations, particularly in the set-up, and starting in 2007 we began

developing new algorithms. The first new developments were using “haikus” to generate the basis, which led to creation/annihilation “hops”, all of which made enormous speed-ups in the set-up.

By this time, Ormand had moved to Lawrence Livermore National Laboratory and Johnson had left for San Diego State University. Working first with a student (Hai Ah Nam) and later a postdoc (Plamen Krastev) at San Diego State University, we carefully studied bottlenecks in parallelization in the application of the Hamiltonian. These studies led us to break up the application of the Hamiltonian by basis sectors, which had two useful outcomes. First, we rewrote our central application routines using simple arrays rather than the derived types used in **REDSTICK**; this gave a speed-up of nearly a factor of 2. Second, applying the Hamiltonian by quantum numbers allowed a more transparent factorization of the Hamiltonian and better parallelization.

With these improvements and dramatic speed-ups, we had an entirely new code, **BIGSTICK**.

Starting around 2014, through the good graces of Wick Haxton we teamed up with UC Berkeley and Lawrence Berkeley Laboratory, and especially Haxton’s graduate student Ken McElvain. Ken’s background in the computer industry proved invaluable, and he was able to tweak the existing code into fantastic performance, especially with regards to parallelism. Hongzhang Shan of Lawrence Berkeley wrote an improved algorithm for using OpenMP in matvec operations.

In addition to the help of Hai Ah Nam and Plamen Krastev, we would also like to thank Esmond Ng, Chao Yang, and Sam Williams, of Lawrence Berkeley National Laboratory, James Vary and Pieter Maris of Iowa State University, and many other colleagues who have provided helpful discussions, suggestions, feedback and insight over the years.

Over the years our primary research funding has come through the U.S. Department of Energy, which has directly and often indirectly supported the development of **BIGSTICK**. We are deeply grateful for this support. Support for this project came primarily from the U.S. Department of Energy, in the form of grants Grant DE-FG02-96ER40985, DE-FG52-03NA00082, DE-FG02-03ER41272, as well as Louisiana State University, Lawrence Livermore National Laboratory, San Diego State University, University of California, Berkeley, and Lawrence Berkeley National Laboratory.

Chapter 2

How we solve the many-body problem

In this chapter we discuss the principles of configuration-interaction (CI) many-body calculations, including some different classes of CI codes, and give an overview of its application in **BIGSTICK**. Configuration-interaction is sometimes called the interacting shell model, as (a) one typically builds the many-body basis from spherical shell-model single particle states and (b) to distinguish from the non-interacting shell model, sometimes also called the independent particle model.

The key points here are:

- We represent the many-body Schrödinger equation as a matrix eigenvalue problem, typically with very large basis dimensions. **BIGSTICK** can compute problems with dimensions up to $\sim 10^7$ on a laptop, up to $\sim 10^8$ on a desktop machine, and up to $\sim 10^{10}$ on parallel supercomputers.
- The large-basis-dimension eigenvalue problem has two computational barriers. The first is how to solve the eigenvalue problem itself, especially given that we almost never need *all* of the eigenvalues. The second is, despite the fact the matrix is typically very sparse, the amount of data required is still huge.
- We address the first problem by using the Lanczos algorithm, which efficiently yields the low-lying eigenpairs.
- We address the second by not explicitly storing all the non-zero matrix elements, but instead invoking a on-the-fly algorithm. This on-the-fly algorithm, first implemented in the Strasbourg group's code **ANTOINE**, exploits the fact that the interaction only acts on two- or three- particles at a time. The on-the-fly algorithm can be thought of as partially looping over spectator particles.

- The on-the-fly algorithm explicitly depends upon the existence of two species of particles, for example protons and neutrons, or in the case of atoms, spin-up and spin-down electrons, so that both the many-body basis and the action of the Hamiltonian can be factorized into two components. This factorization is guided by additive/multiplicative quantum numbers, such as M , the z -component of angular momentum, and parity. This factorization efficiently and losslessly “compresses” information; we outline the basic concepts below, and direct the curious to Johnson et al. [2013] for details.
- In order to implement many-body truncations, we have an additional additive pseudo-quantum number, which we call W . This allows a general, though not infinitely flexible, ability to truncate the basis. We discuss these truncations below, but include for example n -particle, n -hole truncations and the N_{\max} truncation typical of the no-core shell model.

With these efficiencies we can run both “phenomenological” and *ab initio* or no-core shell model calculations, on machines ranging from laptops to supercomputers. Although we do not discuss it in depth in this document, we rely heavily upon both factorization and use of quantum numbers in parallelization.

2.1 Matrix formulation of the Schrödinger equation

The basic goal is to solve the non-relativistic many-body Schrödinger equation for A identical fermions of mass M ,

$$\left(\sum_{i=1}^A -\frac{\nabla_i^2}{2M} + \sum_{i<j} V(\vec{r}_i - \vec{r}_j) \right) \Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_A) = E \Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_A), \quad (2.1)$$

which we often will write using the more compact Dirac bra-ket notation

$$\hat{H}|\Psi\rangle = E|\Psi\rangle. \quad (2.2)$$

Already even Eq. (2.1) is simplified, as it leaves out explicit spin degrees of freedom, and the potential here is purely local and two-body. **BIGSTICK** can handle nonlocal interactions without blinking. **BIGSTICK** can also use three-body forces, although the latter ups computational demands by nearly two orders of magnitude, and in the current release the three-body forces are not optimized.

The basic idea of configuration interaction is to expand the wavefunction in some convenient many-body basis $\{|\alpha\rangle\}$:

$$|\Psi\rangle = \sum_{\alpha} c_{\alpha} |\alpha\rangle \quad (2.3)$$

Then, if the basis states are orthonormal, $\langle\alpha|\beta\rangle = \delta_{\alpha,\beta}$, the Schrödinger equation becomes a matrix eigenvalue equation

$$\sum_{\beta} H_{\alpha,\beta} c_{\beta} = E c_{\alpha}. \quad (2.4)$$

Because we typically deal with many-fermion systems, the wavefunction $|\Psi\rangle$ is completely antisymmetric under interchange of any two particles,

$$\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_i, \dots, \vec{r}_j, \dots) = -\Psi(\vec{r}_1, \vec{r}_2, \dots, \vec{r}_j, \dots, \vec{r}_i, \dots). \quad (2.5)$$

(One can use configuration-interaction methods for many-boson systems, but then the basis states would be totally symmetric, and a completely separate code would be required.) A useful many-body basis are therefore Slater determinants, which are antisymmetrized products of single-particle wavefunctions. (As we will note several times in this manual, it is often important to distinguish between *single-particle* states and *many-body* states, as well as between, for example, *two-body* matrix elements and *many-body* matrix elements.)

We do not explicitly use Slater *determinants* but rather the *occupation representation* of Slater determinants using fermion creation and annihilation operators, also known as second quantization. We assume the reader is comfortable with Slater determinants and the algebra of fermion operators, and therefore give only a terse exposition in order to be clear about our terminology.

Suppose we have a set of N_s single-particle states, $\phi_i(\vec{r})$ where i describes each unique state by its quantum numbers. **BIGSTICK** assumes single-particle states with rotational symmetry, and the available quantum numbers are n , l , j , and m . Here l is the orbital angular momentum, j is the total angular momentum, and m is the z component of total angular momentum. n is the ‘radial’ quantum number; it distinguishes different states with the same angular momentum quantum numbers but a different radial wavefunction. It plays no other internal role in **BIGSTICK**, though it is relevant to calculating the value of matrix element input into the code. Hence **BIGSTICK** can use single-particle states with arbitrary radial components, as long as they are orthonormal; it is up to the user to keep track of what radial wavefunction is being assumed. In many cases, for example in so-called no-core shell model (NCSM) calculations, one uses a harmonic oscillator basis, but that is by no means mandatory or assumed in **BIGSTICK**. In the same way, l really only serves to give the parity of each single-particle state.

BIGSTICK’s efficiency arises from using factorization of the basis, described in the next chapter, which assumes two species of fermions: protons and neutrons in the case of nuclei, or spin-up and spin-down in the case of either the electronic structure of atoms or of fermionic atoms in a spherically symmetric trap. In the former, the total angular momentum $j = |l \pm 1/2|$; in the latter one has ‘ j ’ = l .

BIGSTICK organizes all its work around quantum numbers. In addition to the quantum numbers above, **BIGSTICK** uses another, pseudo-quantum number, weighting, symbolized by w for single-particle states and $W = w_1 + w_2 + \dots$ for many-body states. This pseudo-quantum number is used for many-body

truncations; its use is described in more detail in section 4.2.2, but, briefly, one establishes a maximum W limit on many-body states. In the NCSM one uses $w = N$, the principal harmonic oscillator quantum number, i.e., $2n+l$, and then establishing a maximum sum W yields the important N_{max} truncation which allows for exact elimination of spurious center-of-mass motion.

Starting with a fermion vacuum state $|0\rangle$, the operator \hat{a}_i^\dagger creates the single fermion state ϕ_i . Then the many-body state

$$\hat{a}_{i_1}^\dagger \hat{a}_{i_2}^\dagger \hat{a}_{i_3}^\dagger \dots |0\rangle \quad (2.6)$$

is the occupation representation of the Slater determinant of the single particle states $\phi_{i_1}, \phi_{i_2}, \dots$. For succinctness we will refer to such many-body states as ‘Slater determinants’ even when we mean the occupation representation.

Using a one-body operator such as the kinetic energy \hat{T} can be written using second quantization:

$$\hat{T} = \sum_{ij} T_{ij} \hat{a}_i^\dagger \hat{a}_j = \int \phi_i^* \hat{T} \phi_j \quad (2.7)$$

where $T_{ij} = \langle i | \hat{T} | j \rangle$ is the one-body matrix element of the operator; the actual value is determined through the integral sketched above. Two-body operators, e.g. interactions between two particles, can be similarly represented, though with two annihilation operators followed by two creation operators. It is useful to note that all **BIGSTICK** and similar CI codes read in are the numerical values of the matrix elements. This means the actual form of the single-particle wavefunctions is hidden (although **BIGSTICK**, like nearly all other nuclear CI codes, requires the single-particle states to have good angular momentum, which is to say they transform like spherical tensors [Edmonds, 1996]; this means the only freedom is in the radial wavefunctions. The choice of radial wavefunction affects the numerical value of the input matrix elements, and thus the output of **BIGSTICK**, but operationally **BIGSTICK** is indifferent or even agnostic with respect to those radial wavefunctions. It is the duty and the burden of the user to ‘know’ the radial wavefunctions, especially if calculating any observables besides energy.

The many-body matrix elements are thus exercises in fermion second quantization algebra: $H_{\alpha\beta} = \langle \alpha | \hat{H} | \beta \rangle$ where the basis states $|\alpha\rangle, |\beta\rangle$ and the Hamiltonian operator \hat{H} are all expressed using creation and annihilation operators, given exactly in Appendix B.

In this framework, all CI many-body codes follow the same general process:

1. Read in single-particle basis information. (In principle, as the single-particle space becomes infinite, the solution becomes exact. Due to finite computational resources, of course, we have to truncate the space. Choosing the right space is the heart of the art of many-body calculations.)
2. Construct the many-body basis. Exactly how this is done is key to all CI codes, including **BIGSTICK**.
3. Read in interaction information, in the form of one-, two-, (three-)body information.

4. Construct the many-body matrix elements. Again, exactly how this is done is key to all CI codes, including **BIGSTICK**.
5. Solve the matrix eigenvalue problem (2.4).

2.2 Representation of the basis

The occupation representation is a natural one for the computer as a single particle state can either be occupied or unoccupied, represented by a 1 or a 0. Thus the state

$$\hat{a}_2^\dagger \hat{a}_4^\dagger \hat{a}_5^\dagger \hat{a}_8^\dagger |0\rangle$$

can be represented by the bit string

$$01011001$$

as the single particle states 2,4, 5 and 8 are occupied and the rest unoccupied. Of course, consistency in ordering is important as one has to pay careful attention to phases.

But in the 1970s Whitehead and collaborators used bit manipulation for fast calculation of matrix elements in the occupation scheme [Whitehead et al., 1977]. The basic idea is simple: consider a creation operator, say \hat{a}_4^\dagger , applied to some Slater determinant represented by a bit string. If the 4th bit is 0, then the action of applying \hat{a}_4^\dagger is to create a 1 in its place:

$$\hat{a}_3^\dagger |110001\rangle = |110101\rangle,$$

while if it is already occupied, then the state vanishes in a puff of digital smoke:

$$\hat{a}_3^\dagger |100101\rangle = 0.$$

Similarly an annihilation operator such as \hat{a}_2 will destroy a state if the second bit is empty

$$\hat{a}_2 |100111\rangle = 0,$$

but will replace a 1 bit with a 0,

$$\hat{a}_2 |110101\rangle = -|100101\rangle.$$

The minus sign arises, of course, from fermion anticommutation relations. In this way one can almost trivially find the action of, say, a two-body operator on a state:

$$\hat{a}_3^\dagger \hat{a}_5^\dagger \hat{a}_4 \hat{a}_1 |1101011\rangle = -\hat{a}_3^\dagger \hat{a}_5^\dagger |0100011\rangle = -|01101011\rangle.$$

Then one can search through the basis to find out what state $|01101011\rangle$ is.

In general we work with Hamiltonians which are rotationally invariant, that is, they commute with the generators $\hat{J}_x, \hat{J}_y, \hat{J}_z$ of rotation. This means one can find eigenstates of the Hamiltonian which are also simultaneous eigenstates of

total angular momentum $\hat{J}^2 = \hat{J}_x^2 + \hat{J}_y^2 + \hat{J}_z^2$ and of (by convention) \hat{J}_z , that is,

$$\begin{aligned}\hat{H}|\Psi\rangle &= E|\Psi\rangle; \\ \hat{J}^2|\Psi\rangle &= \hbar^2 J(J+1)|\Psi\rangle; \\ \hat{J}_z|\Psi\rangle &= \hbar M|\Psi\rangle.\end{aligned}$$

We say such states ‘have good angular momentum.’ It is important to note that E generally depends upon J , that is, except for special cases (usually involving additional symmetries) states with different J are not degenerate, for a given value of J the value of E does not depend upon M . In practical terms, what this means is that the Hamiltonian is block-diagonal in J ; it is also block-diagonal in M , but the blocks for the same J but different M have the exact same eigenvalues.

Of course, whether or not the Hamiltonian is explicitly block diagonal depends upon the choice of basis. We call these different choices basis ‘schemes.’

BIGSTICK, like most nuclear CI codes, constructs its many-body basis states using single-particle states which also have good angular momentum, i.e., have eigenvalues $j(j+1)$ and m for \hat{J}^2 and \hat{J}_z , respectively. (Here and hereafter we set $\hbar = 1$.) The addition of total angular momentum is nontrivial, requiring Clebsch-Gordan coefficient, but as \hat{J}_z is the generator of an Abelian subgroup, any product of single-particle states each with good m_i has good total $M = m_1 + m_2 + \dots$

What this means is it is both possible and easy to construct individual Slater determinants which have good M (i.e., are eigenstates of \hat{J}_z). These will almost never be states also of good J . But because \hat{H} commutes with both \hat{J}_z and \hat{J}^2 , if we take all states of a given M and diagonalize \hat{H} , the eigenstates will be guaranteed to also have good J (barring ‘accidental’ degeneracies that rarely occur). Taking states of fixed M is called an *M-scheme basis*. It is the simplest shell-model basis.

But the *M-scheme* isn’t the only choice. As mentioned above, one can also make the many-body Hamiltonian matrix explicitly diagonal in J as well as M . This is a *J-scheme basis*. Such bases are significantly smaller in dimension, typically an order of magnitude smaller than the *M-scheme*. Of course, there are obvious costs. Almost always a state with good J must ultimately be a superposition of *M-scheme* Slater determinants. This means both the *J-scheme* basis states, and the many-body matrix elements in this basis, are more costly to calculate.

(Historically, in chemical and atomic physics one used *configuration state functions* with good angular momentum, which we would call the *J-scheme*. The use of simple Slater determinants in chemical and atomic physics seems to have been introduced by Knowles and Handy [1984] apparently unaware of Whitehead’s innovation.)

One can go even further. Many nuclei exhibit strong rotational bands, which can be reproduced using the group $SU(3)$. If the nuclear Hamiltonian commuted with the Casimir operators of $SU(3)$, or nearly so, then the Hamiltonian would

be block diagonal in the irreps of $SU(3)$, or nearly so, and $SU(3)$ would be *dynamical symmetry*. One can imagine other group structures as well.

For realistic interactions, $SU(3)$ is *not* a good symmetry; among other things, the spin-orbit and pairing interactions strongly mix $SU(3)$ interactions. Nonetheless, detailed calculations show $SU(3)$ is a *quasi-dynamical symmetry*, meaning states in bands show similar amplitudes in the $SU(3)$ irreps.

Because of this, some groups use group-theoretical bases, such as a $SU(3)$ -scheme basis, based upon calculations which suggest that nuclear wavefunction are dominated by a few group irreps. The $SU(3)$ -scheme is just like the J only more so: the basis is more compact, but the basis states and the many-body matrix elements even more complicated to derive. On the other hand, the $SU(3)$ -scheme makes the origin of rotational motion more transparent and potentially offers a more compact representation and understanding of the wavefunctions. Each of these schemes offer advantages and disadvantages.

2.2.1 Use of quantum numbers: factorization of the basis

One of the advantages of the M -scheme is that despite the fact it is the least compact of basis schemes, it can be represented very efficiently with factorization. Factorization is an idea used throughout BIGSTICK, and is most easily illustrated in the basis. More details can be found in Johnson et al. [2013] and references therein.

We work in the M -scheme, which means every many-body basis state has the same definite value of M . If we have an even number of particles, M is an integer, while for odd numbers it will be a half-integer ($1/2, 3/2, -5/2$, etc.). Internally BIGSTICK doubles these so they can be represented by even or odd integers, respectively.

Each basis state, however, is a simple tensor product of a proton Slater determinant and a neutron Slater determinant. Because the m quantum numbers are additive, we have the total $M = M_p + M_n$, the sum of proton and neutron M -values.

Absent other constraints, *every* proton Slater determinant with M_p not only can but *must* be combined with *every* neutron Slater determinant with $M_n = M - M_p$; this, in part, guarantees that rotational invariance is respected and that the final eigenstates will have good total J . This in turn leads to a shortcut.

Consider the case of the ^{27}Al nucleus, using the sd valence space. This assumes five valence protons and six valence neutrons above a frozen ^{16}O core. The total dimension of the many-body space is 80,115, but this is constructed using only 792 five-proton states and 923 six-neutron states.

The reader will note that $792 \times 923 = 731016 \gg 80115$. Indeed, not every five-proton state can be combined with every six-neutron state. The restriction is due to conserving certain additive quantum numbers, and this restriction turns out to limit usefully the nonzero matrix elements of the many-body Hamiltonian, which we will discuss more in the next section.

For our example, we chose total $M = +1/2$ (though we could have chosen a different half-integer value). This basis requires that $M_p + M_n = M$; and for

Table 2.1: Decomposition of the M -scheme basis for 5 protons and 6 neutrons in the sd valence space (^{27}Al), with total $M = M_p + M_n + 1/2$. Here “pSD” = proton Slater determinant and “nSD” = neutron Slater determinant, while “combined” refers to the combined proton+neutron many-body basis states. The subset of the basis labeled by fixed M_p (and thus fixed M_n) we label a ‘sector’ of the basis.

M_p	# pSDs	M_n	# nSDs	# combined
+13/2	3	-6	9	27
+11/2	11	-5	21	231
+9/2	28	-4	47	1316
+7/2	51	-3	76	3876
+5/2	80	-2	109	8720
+3/2	104	-1	128	13,312
+1/2	119	0	142	16,898
-1/2	119	+1	128	15,232
-3/2	104	+2	109	11,336
-5/2	80	+3	76	6080
-7/2	51	+4	47	2444
-9/2	28	+5	21	588
-11/2	11	+6	9	99
-13/2	3	+7	1	3
Total	792		923	80,115

some given M_p , *every* proton Slater determinant with that M_p combines with *every* neutron Slater determinant with $M_n = M - M_p$. This is illustrated in Table 2.1, which shows how the many-body basis is constructed from 792 proton Slater determinants and 923 neutron Slater determinants. Note we are “missing” a neutron Slater determinant; the lone $M_n = -7$ state has no matching (or ‘conjugate’) proton Slater determinants.

As a point of terminology, we divide up the basis (and thus any wavefunction vectors) into *sectors*, each of which is labeled by M_p , and any additional quantum numbers such as parity Π_p ; that is, all the basis states constructed with the same M_p (Π_p , etc.) belong to the same basis ‘sector’ and have contiguous indices. Basis sectors are also useful for grouping operations of the Hamiltonian, as described below, and can be the basis for distributing vectors across many processors, although because sectors are of different sizes this creates nontrivial issues for load balancing.

While we can represent the 80,115 basis states of ^{27}Al in the sd with 792 proton Slater determinants and 923 neutron Slater determinants, the storage is even more impressive for large systems. For example, in the pf shell, ^{60}Zn , with 10 valence protons and 10 valence neutrons, has for $M = 0$ a basis dimension of 2.3 *billion*. But these are represented by $\sim 185,000$ proton Slater determinants and the same number of neutron Slater determinants. (In principle with self-conjugate systems $N = Z$ systems one could gain further savings by keeping

only one set of Slater determinants. Because that is a small number of nuclides, we chose not to do so.) The savings are not as dramatic for no-core shell model calculations with N_{max} truncation. For example, ^{12}C in a basis of $N_{\text{max}} = 10$ has a basis dimension of 7.8 billion, constructed from 1.8 million each proton and neutron Slater determinants. The reason for the lessened efficiency is the many-body truncation.

We note that factorization not only provides dramatic lossless compression of data, it also accelerates the set up of data. In the set up phase of any CI code, one of the major tasks is searching through long series of bitstrings and, when one uses quantum numbers to organize the data, sorting. Factorization improves this by reducing the length of lists to be searched and sorted. Our second level of factorization further reduces those lists, making searches and sorts even faster.

While factorization of the Hamiltonian was, to the best of our knowledge, pioneered by Caurier and Nowacki [1999] in the code ANTOINE and adopted as well by EICODE [Nuclear Shell Model], NuShell/NuShellX [Brown and Rae, 2014], and KSHELL [Shimizu, 2013] (and possibly others we are unaware of), BIGSTICK has uniquely implemented a second level of factorization. Because most users never see this level, we direct those interested to our paper for more details.

BIGSTICK does provide some information about this. In normal runs, as well as in modeling runs, you will see

```
.... Building basis ...

Information about basis:
there are          27  sectors          1
there are          27  sectors          2
                38760  SDs for species      1
                184722 SDs for species      2
Total basis =                501113392

.... Basis built ...
```

The above example is for ^{56}Fe in the pf shell with $M = 0$. The *sectors* are the subsets of the proton and neutron Slater determinants (‘SDs’) with fixed quantum number M , parity, and optionally W . Here ‘species 1’ refers to protons and ‘species 2’ refers to neutrons.

2.3 The Lanczos algorithm and computational cost

With bit manipulation allowing one to quickly calculate matrix elements, one could address much larger spaces, spaces so large they were not amenable to complete diagonalization, e.g., through the Householder algorithm [Parlett,

1980, Press et al., 1992]. But in nuclear structure configuration-interaction one almost never wants all the eigensolutions; instead one typically just wants the low-lying states. Thus Whitehead et al. [1977] introduced another innovation: use of the Lanczos algorithm to find the extremal eigenstates.

The Lanczos algorithm is a subspecies of Arnoldi algorithms. We describe the Lanczos algorithm in Chapter 7, but the key idea is that, starting from an initial vector often called the *pivot*, one iteratively constructs a sequence of orthonormal basis vectors that form a *Krylov subspace*, as well as the elements of the Hamiltonian in that subspace. The genius of Arnoldi/Lanczos algorithms is that they use the matrix to be diagonalized to construct the basis vectors; by applying the Hamiltonian matrix to a given basis vector one constructs, after orthogonalization, the next basis vector. One can show via the classical theory of moments that the extremal eigenvalues of the Hamiltonian in the Krylov subspace quickly converge to those of the full space. Although it depends upon the model space, the Hamiltonian, and the choice of pivot (starting vector to kick off the Lanczos iterations), one can often reach a converged ground state energy in as few as twenty Lanczos iterations, and the lowest five states in as few as 100 iterations.

As we’ve seen, in the configuration-interaction method, the many-body Schrödinger equation has been recast as a matrix eigenvalue equation (2.4), which in turn is generally solved by some iterative Arnoldi method such as the Lanczos algorithm. In all of those algorithm there are two fundamental operations: matrix-vector (matvec) multiplication (or, in the case of block-Arnoldi methods, matrix-matrix multiplication), and linear algebra among those vectors, i.e., dot products, normalization, orthogonal subtraction, and the like.

Now let us think about the computational cost of carrying out CI, both in terms of operations (time) and memory (storage). Before doing so let us highlight a key point. Most often in discussing CI one cites the basis dimension. But, as we will argue below, the real measure of the computational cost is the number of nonzero matrix elements. Now, for any given scheme, the number of nonzero matrix elements scales with the basis dimension. However, for different schemes the proportionality is different: *J*-scheme is denser than *M*-scheme; furthermore, even within the same basis scheme, different truncations have different densities, e.g., the NCSM is much denser than ‘phenomenological’ calculations. Therefore, for absolute comparison of the computational cost of a problem, the number of nonzero matrix elements is a much better measure than basis dimension.

That said, let us look at the computational cost of matvec:

$$w_\alpha = \sum_\beta H_{\alpha,\beta} v_\beta.$$

Let the dimension of the vector space be N . If the many-body matrix \mathbf{H} is a fully dense (but real, symmetric) matrix, the above matvec requires N^2 operations as well as storage of N^2 many-body matrix elements. However, \mathbf{H} is almost never fully dense. This can be most easily understood in the *M*-scheme,

where the fundamental occupation-space basis states can be represented as raw bit strings. A two-body interaction can at most shift two bits. Therefore if two basis states $|\alpha\rangle$ and $|\beta\rangle$ differ by more than two bits, the matrix element between them *must* be zero. A typical ‘sparsity’ of M -scheme Hamiltonians is 2×10^{-6} , that is, only two out of every million many-body matrix elements is nonzero. (Three-body forces, naturally, lead to denser matrices, roughly two orders of magnitude denser.)

If one has a basis dimension of a million, then there are roughly a million nonzero matrix elements; because one needs not only the value of the matrix element but some index to local it in the matrix, in single precision this requires roughly 8 megabytes of memory. When one goes up to a basis dimension of one billion, however, this number goes up by $(10^3)^2$ to 8 terabytes! Reading 8 Tb of data even from fast solid state disks is a very slow proposition. If one stores the matrix elements in core memory across many processors, as the code `MFDn` does, this requires a minimum of many hundreds if not thousands of processors.

2.4 Representation of the Hamiltonian

As discussed above, M -scheme configuration-interaction calculations require a many-body vector space of very large dimensions, and the many-body Hamiltonian matrix, while very sparse, still in large cases nonetheless the nonzero matrix elements end requiring a huge amount of data.

If you were to examine closely, say, the bit representation of the basis states, or the nonzero matrix elements, you’d find something confounding: quite a lot of data is repeated, over and over. The same proton bit strings (which we generally call proton Slater determinants, although technically they are *representations* of said determinants) are repeated many times, sometimes many millions of times or more, and the same for the neutron bit strings (neutron Slater determinants). In the same way, the same values appear, thousand and millions of times, in the non-zero many-body matrix elements, though with both positive and negative values.

This redundancy can not only be understood, it can be turned to our advantage through *factorization*, both of the basis and of matvec operations.

While the curious reader should look to our paper Johnson et al. [2013] for details, the basic idea is as follows. Any two-body Hamiltonian can be split into forces that act only on protons, forces that act only on neutrons, and interactions between protons and neutrons. Consider forces acting only on protons; in a factorized basis, the neutrons are spectators. If we write our basis states as a simple tensor product between a proton ‘Slater determinant’ $|i_p\rangle$ and a neutron Slater determinant, $|j_n\rangle$, so that the basis state $|\alpha\rangle = |i_p\rangle|j_n\rangle$ the pure proton Hamiltonian matrix element is

$$\langle\alpha|\hat{H}_{pp}|\alpha'\rangle = \langle i_p|\hat{H}_{pp}|i'_p\rangle\delta_{j_n,j'_n}. \quad (2.8)$$

We therefore only have to store the proton matrix element $\langle i_p|\hat{H}_{pp}|i'_p\rangle$, and can trivially loop over the neutron Slater determinants. You can see how you could

Table 2.2: Number of one- and two-body ‘jumps’ and storage requirements for representative atomic nuclei in different model spaces (described in Appendix B). For storage of nonzero matrix elements (penultimate column) we assume each many-body matrix element is stored by a 4-byte real number and its location encoded by a single 4-byte integer. Storage of a single jump (initial and final Slater determinants for a species, and matrix element and phase) requires 13 bytes. All storage (final two columns) are in gigabytes (GB).

Nuclide	space	basis dim	# 1-body jumps	# 2-body jumps	Store m.e.s	Store jumps
^{28}Si	sd	9.4×10^4	4.8×10^4	7.6×10^3	0.2	0.002
^{52}Fe	pf	1.1×10^8	4.0×10^6	8.5×10^6	700	0.16
^{56}Ni	pf	1.1×10^9	1.5×10^7	4.0×10^7	9800	0.6
^4He	$N_{\max} = 22$	9×10^7	5.3×10^8	4.7×10^9	9300	69
^{12}C	$N_{\max} = 8$	6×10^8	6×10^8	3×10^9	5200	45
^{13}C	$N_{\max} = 6$	3.8×10^7	7×10^7	3×10^8	210	4.3

get dozens, hundreds, or thousands of matrix elements with the same value, just with different j_n . Furthermore, because the neutron Slater determinants are frozen, the quantum numbers cannot change, which severely restricts the action of the proton-only Hamiltonian. The matrix elements $\langle i_p | \hat{H}_{pp} | i'_p \rangle$ are called *jumps* and we only need to store them (and know of the neutron indices j_n over which to loop).

For proton-neutron interactions the action is more complicated but the same basic ideas hold: one stores separate proton jumps and neutron jumps and reconstruct the value of the matrix elements. Table 2.2 shows the storage for nonzero matrix elements and of jumps needed for a number of representative nuclei, in both phenomenological and NCSM calculations. You can see there are at least two orders of magnitude difference. Thus, for example, ^{52}Fe , which would require 700 Gb of storage for just the nonzero matrix elements, only needs less than a Gb for storage in factorization (in this particular case, storage of the Lanczos vectors is much higher burden) and thus can be run on an ordinary desktop computer.

The price one pays, of course, is the factorized reconstruct-on-the-fly algorithm is much more complicated.

2.5 An incomplete survey of other codes

While this manual is about **BIGSTICK**, it is appropriate to put it in the context of other (nuclear) configuration-interaction codes. One can broadly classify them by (a) basis scheme, (b) representation and storage of many-body matrix elements, (c) rank of interactions (i.e., two-body only or two- and three-body forces), (d) parallelism, if any, and finally (e) general area of applicability, e.g., primarily to phenomenological spaces, which usually means a frozen core, and interactions, or to *ab initio* no-core shell model calculations. Please keep in

mind that most of these codes are unpublished or have only partial information published, and that many of the details have been gleaned from private conversations; information on some codes, such as the powerful Japanese code **MSHELL**, do not seem to be available. We apologize for any accidental misrepresentations. All of these codes have powerful capabilities and have made and are making significant contributions to many-body physics.

Among the very earliest codes was the Oak Ridge-Rochester code from the 1950s and 1960s, which fully diagonalized the Hamiltonian after computing the many-body J -scheme matrix elements via coefficients of fractional parentage. It was succeeded by the Whitehead (Glasgow) code and its descendents, which used bit manipulation to compute the many-body matrix elements in the M -scheme, and solved for low-lying eigenstates using the Lanczos algorithm, **ANTOINE** [Caurier and Nowacki, 1999], **MFDn** [Sternberg et al., 2008], and **KSHELL** [Shimizu, 2013] are also M -scheme codes. Examples of J -scheme codes in nuclear physics include **AXBASH** [Brown et al., 1985] and its successors **NuShell** and **NuShellX** [Brown and Rae, 2014], **NATHAN** [Caurier et al., 1999], and **EICODE** [Nuclear Shell Model, Toivanen, 2006].

There have been attempts to use group theory to construct so-called symmetry-guided bases. The main effort is in $SU(3)$. While originally applications were to phenomenological calculations, recently the strongest application of $SU(3)$ has been in *ab initio* calculations. One barrier to standard no-core shell model calculations in the j - j scheme is that the harmonic oscillator basis allow for exact projection of spurious center-of-mass motion but do poorly in describing the exponential tail of the wavefunction; brute force inclusion of many oscillator shells leads to a crippling explosion in dimensions. By contrast, the multi-shell generalization of $SU(3)$, the symplectic group $Sp(3, R)$, naturally couples to higher shells while still projecting spurious states, leading to better radial wave functions and description of, for example, electromagnetic interactions Escher and Draayer [1999]. Recent work has demonstrated $Sp(3, R)$ is a strong quasidynamical symmetry arising naturally in *ab initio* calculations Dytrych et al. [2007b,a]. Success in applying this insight to structure calculations [Dytrych et al., 2008a,b, Launey et al., 2014] did not use symplectic symmetry directly, but instead used a tower of $SU(3)$ irreps, built in different harmonic oscillator shells. Such work opens up the possibility of carrying out calculations not currently possible in the standard j - j coupled NCSM, such as describing the Hoyle state in ^{12}C , known to require many excitations in a harmonic oscillator space Neff [2012]. Although this approach is very promising, only time will tell for sure if the advantages gained by group theory will outweigh the technical difficulties needed to implement, for although the bases are small, they are significantly denser, and furthermore the group theory is very challenging.

Regarding access to the many-body Hamiltonian matrix element, the Oak Ridge-Rochester and Whitehead codes stored matrix elements on disk, as do **AXBASH** and **NuShell**. The very successful code **MFDn** [Sternberg et al., 2008], primarily but not exclusively used for no-core shell model calculations, stores the many-body Hamiltonian matrix elements in RAM; which is much faster than storing on disk, but for all but the most modest of problems requires distribution

across hundreds or thousands of nodes on a parallel computer spread across many MPI processes.

Factorization methods, pioneered in **ANTOINE** [Caurier and Nowacki, 1999], have been used in several other major CI codes: **NATHAN** [Caurier et al., 1999], **EICODE** [Nuclear Shell Model, Toivanen, 2006], **NuShellX** [Brown and Rae, 2014], **KSHELL** [Shimizu, 2013], and our own unpublished codes **REDSTICK** (so named because it was originated at Louisiana State University, located in Baton Rouge), and of course **BIGSTICK**. Methods similar to factorization have been used in quantum chemistry (see Ref. ?? and references therein), which allowed quantum chemists to reach dimensions of over a billion (but again: the computational difficulty is not only the dimensionality of the vectors but also the number of nonzero matrix elements). Factorization has also been used in nuclear structure physics as a gateway to approximation schemes ?????. The codes most widely used by people beyond their authors have been **AXBASH** and its successor **NuShell/NuShellX**, and **ANTOINE**. Because of their wide use, and because one of us (Ormand) heavily used **AXBASH**, the default formats for our input `.sps` and `.int` files are heavily modeled upon the **AXBASH/NuShell/NuShellX** formats.

Like **BIGSTICK**, **MFDn** has been parallelized with both MPI and OpenMP and has carried out some of the largest supercomputer runs in the field. **NuShellX** has only OpenMP parallelization. The parallelization of other codes is unknown.

In closing, we note that besides configuration-interaction there are many other approaches to the many-body problem, such as the coupled cluster method, the Green’s function Monte Carlo method, the in-medium similarity renormalization group, density functional methods, and so on, each with their own advantages and disadvantages. There are also methods closely related to configuration interaction, such as the ‘Monte Carlo shell model,’ the ‘shell-model Monte Carlo,’ generator-coordinate codes, and the importance truncation shell model. The main weakness of configuration interaction is that it is not size extensive, which means unlinked diagrams must be cancelled and thus the dimensionality of the problem grows exponentially with particle number and/or single-particle basis. The advantages of CI is: it is fully microscopic; its connection to the many-body Schrödinger equation (2.1) is pedagogically transparent; it generates excited states as easily as it does the ground state; it can handle even and odd numbers of particle equally well and works well far from closed shells; and finally places no restriction on the form of either the single-particle basis or on the interaction (i.e., local and nonlocal forces are handled equally well, because the occupation space basis is intrinsically nonlocal to begin with).

Chapter 3

Getting started with BIGSTICK

BIGSTICK is a configuration-interaction many-fermion code, written in Fortran 90. It solves for low-lying eigenvalues of the Hamiltonian of a many fermion system; it does this by creating a basis of many-body states of Slater determinants. The Hamiltonian is assumed to be rotationally invariant and to conserve parity, and is limited to two-body (and three-body, in progress) forces. Otherwise no assumptions are made about the form of the single-particle states or of the Hamiltonian.

BIGSTICK allows for two species of fermions, typically protons and neutrons for nuclear cases. BIGSTICK is flexible, able to work with "no-core" systems and phenomenological valence systems alike, and can compute the electronic structure of single atoms or cold fermionic gases (in which cases the two species are interpreted as "spin-up" and "spin-down"). BIGSTICK has a flexible many-body truncation scheme that covers many common truncations. For nuclei it can assume isospin symmetry or break isospin conservation. Interaction matrix elements must be pre-computed by a third-party program and stored as a file, but BIGSTICK accepts a variety of matrix element formats.

3.1 What can BIGSTICK do?

What capabilities does the BIGSTICK code have? BIGSTICK one can:

- compute the ground state energies and low-lying excitation spectra, including angular momentum and, if relevant, isospin, of many-body systems with a rotationally invariant Hamiltonian; wave functions are also generated;
- compute expectation values of rotationally invariant one- and two-body operators;

- compute one-body densities, including transition densities, among the low-lying levels, which allows one to calculate transition rates, life times, moments, etc.;
- compute transition strength probabilities or strength functions for one-body transition operators, useful when one needs to model transitions to many excited states;
- use the strength function capability to decompose the wave function by the eigenvalues of an operator, such as the Casimir of some group.

Along with this, one can ask, what are **BIGSTICK**'s limitations? This largely depends upon the computer used and the many-body system. In low-energy nuclear structure physics, which is the main focus of our research, one can easily run on a laptop any nuclide in the phenomenological *sd* space, and on a workstation reach most nuclides in the phenomenological *pf* space. Although dimensionality is not the most important determination of computational burden, one can generally run cases of dimension up to a few million or even tens of millions, if one is patient, on a laptop, a few hundred million on a workstation, and a few billion on a parallel supercomputer.

As always, of course, much of the limitations depend upon the user. Although we provide a few example input files, it is generally up to the user to provide files for the model space, the interaction, and codes to postprocess density matrices into transitions. (We do provide some tools for this.)

3.2 Downloading and compiling the code

BIGSTICK was developed for UNIX/Linux/macOSX systems. We made no effort to adapt it to running under Microsoft Windows.

To get **BIGSTICK**, download it from [??] GitHub:

```
git clone https://www.github.com/cwjsdsu/BigstickPublic/
```

Your distribution includes a makefile. We have developed **BIGSTICK** to compile and run successfully with Intel's ifort compiler and GNU gfortran. You may need to edit the makefile to put in the correct compiler and/or if you wish to use for example LAPACK libraries. We have written the code to require minimal special compile flags.

For example,

```
PROMPT> make serial
```

makes a serial version of the code with the Intel ifort compiler by default. Several other options are:

```
PROMPT> make openmp      → an OpenMP parallel version using ifort
PROMPT> make gfortran     → a serial version using gfortran
PROMPT> make gfortran-openmp → an OpenMP version using gfortran
and so on. To see all the options encoded into the makefile,
```

PROMPT> make help

Each of these generates an executable with the nonstandard extension `.x`, chosen to make deletion easy: `bigstick.x`, `bigstick-openmp.x`, `bigstick-mpi.x`, and `bigstick-mp-omp.x`. There are options for compiler on a number of supercomputers. Please keep in mind, however, that compilers and compile flags on supercomputers are a Red Queen’s Race, and it is up to the user to tune the makefile for any given configuration.

Libraries. In routine operations, `BIGSTICK` uses the Lanczos algorithm to reduce the Hamiltonian matrix to a truncated tridiagonal matrix whose eigenvalues approximate the extremal eigenvalues of the full matrix. This requires an eigensolver for the tridiagonal. For modest cases, one can also choose to fully diagonalize the Hamiltonian, using a Householder algorithm. (In practice we find this can be done quickly for basis dimensions up to a few thousand, and with patience can be done up to a basis dimension $\sim 10^4$.) For both cases we use the LAPACK routine `DSYEV`, which solves the real-valued, double-precision symmetric eigenvalue problem. The actual matrix elements are given in single-precision, but we found when the density of eigenvalues is high, double-precision gives us better values for observables, including angular momentum J and isospin T .

Although in principle one could link to a library containing `DSYEV`, in practice this is highly platform dependent. Also, except for special cases where one is fully diagonalizing very large matrices and are impatient, the call to `DSYEV` is a tiny fraction of the time. Hence we supply an unmodified copy of `DSYEV` and required LAPACK routines, and there is no need to call any libraries.

3.3 Required input files

In order to solve the many-body Schrödinger equation, `BIGSTICK` requires at least two inputs:

- (a) A description of the single-particle space, usually through a file with extension `.sps` (although if one is running a no-core shell model calculation, there is an option to generate this automatically); and
- (b) A file containing the matrix elements of the interaction, in the form of single particle energies and two-body matrix elements (and, optionally, three-body matrix elements).

We supply several example cases for both inputs, including some commonly used spaces and interactions. But in general it is the user’s duty to supply these input files and, importantly, to make sure they are consistent with each other, i.e., to make sure the ordering of single-particle orbits in the `.sps` file is consistent with those in the interaction file. We describe the file formats in detail in Chapter 4

3.4 Running the code

BIGSTICK has a simple interactive input. It can also be run by pipelining the input into the code.

To run:

```
PROMPT>bigstick.x
```

(we recommend you keep the source code, the executable, and the input data files in separate directories, and make sure the executable is in your path). We use the nonstandard extension `.x` to denote executables.

First up is a preamble, with the version number, information on parallel processes, and a reminder for citations:

```
BIGSTICK: a CI shell-model code
Version 7.8.1 Aug 2017
```

```
by C. W. Johnson, W. E. Ormand,
K. S. McElvain, and H.Z. Shan
For reference please cite:
C. W. Johnson, W. E. Ormand, and P. G. Krastev
Comp. Phys. Comm. 184, 2761-2774 (2013)
[also found in arXiv:1303.0905]
```

```
This code distributed under Open Source License GPL.v2
```

```
Running on NERSC_HOST: none
scratch_dir (*.wfn,...): .
Number of MPI processors = 1
NUM_THREADS = 1
```

Next and most important is the main menu:

```
* * * * *
*
*           OPTIONS (choose 1)
* (i) Input automatically read from "autoinput.bigstick" file
* (note: autoinput.bigstick file created with each nonauto run)
* (n) Compute spectrum (default); (ns) to suppress eigenvector write up
* (d) Densities: Compute spectrum + all one-body densities
* (dx) Densities: Compute one-body densities from previous run
* (x) eXpectation value of an operator (from previous wfn)
* (?) Print out all options
*
* * * * *
```

Enter choice

The most common choice is “n” for a normal run.

To facilitate batch runs or multiple runs with similar inputs, each time BIGSTICK runs it creates a file `autoinput.bigstick`. This file can be edited; choosing “i” from the initial menu will direct BIGSTICK to read all subsequent commands from that file.

Next up:

Enter output name (enter "none" if none)

If you want your results stored to files, enter something like `Si28run1`.

The code will then create the following files:

`Si28run1.res` : text file of eigenenergies and timing information.

`Si28run1.wfn`: a binary file (not human readable) file of the wavefunctions for post-processing or for other runs, e.g. “x” expectation values, etc.

`Si28run1.log`: a logfile of the run, useful for tracking the exact conditions under which the run happened, as well as diagnosing problems.

Other files generated but not need by most users:

`Si28run1.lcoef` : text file of Lanczos coefficients;

`timinginfo.bigstick` and `timingdata.bigstick`: files on internal timing;

`distodata.bigstick`: a file contain information on distribution of work across MPI processes;

and others used primarily by the authors for diagnosing behavior.

If you enter “none,” the `.bigstick` files will be created but no results file (`.res`) and no wavefunction file (`.wfn`).

Enter file with s.p. orbit information (`.sps`)

(Enter "auto" to autofill s.p. orbit info)

This provides information about the single-particle space. A typical answer might be `sd.sps`, which reads in information about the *sd* valence space, which we describe in more detail below. The `auto` option can only be used for “no-core” nuclear shell-model calculations.

Enter # of protons, neutrons

These are the valence protons and neutrons. So, for example, if one wants to compute ^{24}Mg , which has 12 protons and 12 neutrons, but the *sd* single-particle space assumes a closed ^{16}O core, so one has 4 valence protons and 4 valence neutrons. For other kinds of fermions, see the appendix.

Enter 2 x J_z of system

BIGSTICK is a “M-scheme” code, meaning the many-body basis states have fixed total $M = J_z$ (as opposed to J-scheme codes such as `NuShell` which the basis has fixed total J). You must enter an integer which is twice the desired value of M . If there are an even number of particles, this is usually 0. For an odd number of nucleons, you must enter an odd integer, typically ± 1 . Because the

Hamiltonian is rotationally invariant, the results should not change for a value $\pm M$. One can choose a non-minimal M if, for example, you are interested in high-spin states.

Enter parity +/- :

In addition to fixed M , BIGSTICK has fixed parity. BIGSTICK automatically determines if more than one kind of parity is allowed and asks for the parity. The *sd* space, for example, has only positive parity states, and so this input is automatically skipped.

If you would like to compute both parities, enter '0'. (At the current time, this is necessary if you want to compute parity-changing transitions, as for any transition calculations BIGSTICK must work in the same basis.)

Would you like to truncate ? (y/n)

In some cases it is possible to truncate the many-body space, discussed in detail in section 4.2.2.

BIGSTICK will then generate the basis; in most cases this takes only a fraction of a second. BIGSTICK will print out some information about the basis, which you can generally ignore.

The next item is to read in the matrix elements of the Hamiltonian.

Enter interaction file name (.int)

(Enter END to stop)

You can enter in a number of interaction files. The format for the interaction files will be discussed below.

Enter scaling for spes, A,B,X ((A/B)^X) for TBMEs

(If B or X = 0, then scale by A)

After the interactions files have been read in, BIGSTICK sets up the jump arrays for reconstructing the matrix elements on the fly. After that, the eigensolver menu comes up:

```

/ -----\
|
|   DIAGONALIZATION OPTIONS (choose one)   |
|
| (ex) Exact and full diagonalization (use for small dimensions only) |
|
| (ld) Lanczos with default convergence (STANDARD) |
| (lf) Lanczos with fixed (user-chosen) iterations |
| (lc) Lanczos with user-defined convergence |
|
| (td) Thick-restart Lanczos with default convergence |
| (tf) Thick-restart Lanczos with fixed iterations |

```

```
| (tc) Thick-restart Lanczos with user-defined convergence |
| (tx) Thick-restart Lanczos targeting states near specified energy |
| | |
| (sk) Skip Lanczos (only used for timing set up) |
| | |
\ -----/
```

As noted, the standard choice is ‘ld’ for default Lanczos. Other options are discussed later.

ld

```
Enter nkeep, max # iterations for lanczos
(nkeep = # of states printed out )
```

Except for very small cases, BIGSTICK does not find all the eigenvalues. Instead it uses the Lanczos algorithm (introduced by Whitehead et al to nuclear physics) to find the low-lying eigenstates. The variable *nkeep* is the number of targeted eigenpairs; typical values are 5-10. One can either set a fixed number of iterations, typically 100-300, or set a maximal number of iterations and allow BIGSTICK to stop sooner using a test for convergence (discussed in detail below).

BIGSTICK will then carry out the Lanczos iterations, printing out intermediate eigenvalues. The final result, which if a output file name was chose is also written to the .res file, looks like

State	E	Ex	J	T
1	-149.77950	0.00000	0.000	0.000
2	-147.78011	1.99939	2.000	0.000
3	-144.92743	4.85207	4.000	0.000
4	-144.11148	5.66801	0.000	0.000
5	-142.72124	7.05826	3.000	0.000

This is fairly self-explanatory. *E* is the absolute energy, *Ex* the excitation energy relative to the first state, and *J* and *T* are the total angular momentum and isospin, respectively. Even though only *M* is fixed, because the Hamiltonian commutes with \hat{J}^2 the final states will have good *J*. Lack of good *J* signals most likely signals lack of convergence, or states degenerate in energy but with different *J*. Lack of good *J* can also signal an error in the input file (specifically, a disallowed *J* for a particular set of orbits; we have written error traps to catch such a problem), or, finally, a bug in the code itself.

If the input matrix elements respect isospin, then *T* should also be a good quantum number. BIGSTICK allows one to read in isospin-breaking matrix elements, discussed in more detail in section 4.3.2.

BIGSTICK can also compute one-body density matrix elements at the end of a run; choose option d in the initial menu. The format and conventions for the density matrices are in section 4.4.3.

The wavefunctions are saved to a .wfn file, unless you choose option ns in the initial menu. BIGSTICK can then post-process the files, for example

computing the expectation value of a scalar (Hamiltonian-like) operator, section 4.6.1; compute overlap between wavefunctions from two different runs, section ; or apply a non-scalar transition operator to a wavefunction and then compute the strength distribution of that transition, sections 4.6.2, 5.2.2, and 5.2.3.

3.5 Some sample runs

In the directory `SampleRuns` that should be found in your distribution, you will find various examples of runs, along with sample outputs to check the code is working correctly.

3.6 Typical run times

In this section we survey ‘typical’ run times for calculations using `BIGSTICK`. Of course, these depend upon the clock-speed of your chip as well as the compiler, as well as how much parallelism you are exploiting. As we show below, `BIGSTICK` does scale well in parallel mode.

Table 3.1 gives, for a variety of nuclides, the dimensionality of the space, the number of operations (which is approximately though not exactly the number of nonzero matrix elements), the minimal storage which would be required to store the nonzero matrix elements, and finally an approximate run time, assuming 150 Lanczos iterations on a serial machine. The actual time may vary a lot, depending on clock speed and how efficiently the operations are actually processed. Parallelism, of course, can speed up the wall clock times considerably.

Empirically, one finds that the number of nonzero matrix element (here, operations) generally scales like $(\text{dim})^{1.25}$ for two-body interactions, and $\approx (\text{dim})^{1.5}$ for three-body forces.

Nuclide	space	dim	# ops	min. store	run time
^{24}Mg	<i>sd</i>	28,503	8.6M	34 Mb	5 s
^{48}Cr	<i>pf</i>	2 M	1.5 B	6 Gb	15 min
^{51}Mn	<i>pf</i>	44M	41B	160 Gb	9 hr
^{56}Fe	<i>pf</i>	500M	0.9 T	3.6 Tb	6 d
^{60}Zn	<i>pf</i>	2.3B	5T	20 Tb	35 d
^{12}C	N_{\max} 6	32M	41 B	160 Gb	7 hr
^6Li	N_{\max} 12	49M	180 B	700 Gb	30 hr
^{12}C	N_{\max} 8	594M	1.2T	5 Tb	8 d
^{16}O	N_{\max} 8	1B	2 T	8 Tb	14 d
^{10}B	N_{\max} 10	1.7B	5 T	20 Tb	35 d
^6Li	N_{\max} 16	800M	7T	27 Tb	46 d

Table 3.1: ‘Typical’ run times for various nuclides, running in serial for 150 Lanczos iterations. Here ‘min. store’ is an estimate of the minimal storage required for nonzero matrix elements.

Chapter 4

Using BIGSTICK, in detail

BIGSTICK has two basic modes. It can calculate many-body spectra and wave functions, and it can process those wave functions in several ways. In order to generate the low-lying spectrum and wave functions, you need to, first, define the model space, and second, provide an interaction.

4.1 Overview of input files

BIGSTICK uses three classes of externally generated files. Two mandatory files are files which define the single-particle space, and files for interaction matrix elements. Optionally, BIGSTICK can also use files for one-body transition matrix elements. Here we briefly summarize those files, and in later sections give more details.

Files which define the single-particle space have the extension either `.sps` or `.sp`. When prompted, the user only supplies the name, not the extension, i.e., if the file is `sd.sps` only enter `sd`. BIGSTICK will automatically search for both `sd.sps` and, if not found, then `sd.sp`. These files can assume isospin symmetry or separate proton-neutron orbits, but at this time, BIGSTICK requires that the proton and neutron single-particle spaces initially be the same. BIGSTICK can however truncate the proton and neutron spaces differently.

If the user is carrying out a ‘no-core shell-model’ calculation where the single-particle orbits are assumed to occur in a default order, BIGSTICK has an ‘auto’ option for defining the single-particle space and no input file is required.

BIGSTICK accepts two classes of files for interaction matrix elements. The default format is derived from OXBASH/NuShell. It can be in isospin-conserving format or in explicit proton-neutron format. Be aware that the latter has two possibilities for normalization of the proton-neutron states. These files are used primarily though not exclusively for phenomenological spaces and interactions. All files with this format must end in the extension `.int`, and as with the single-particle files, one enters only the name, i.e., if the file is `usda.int` one enters in only `usda`. If the file is in isospin-conserving format, you only need to enter

the name of the file. If the file is in proton-neutron format, you must first tell BIGSTICK the normalization convention, see section 4.3.2. These files have broad options for scaling the magnitudes of matrix elements, see section 4.3.1.

BIGSTICK also accepts files in a format readable by the MFDn code. Here one must enter in the **full** name of the file, even if it has the extension `.int`, so that if the file is `TBME.int` you enter `TBME.int` not `TBME`; this signals to BIGSTICK the format. Go to section 4.3.3 for more details.

Finally, BIGSTICK can apply a one-body operator to a wave function in order to generate a transition strength function. These have extension `.opme`. These are defined in section 4.6.2, with advanced instruction and examples in sections 5.2.2 and 5.2.3.

While we supply sample files of these various formats, in general it is **the responsibility of the user to generate or obtain input files**.

All other files BIGSTICK needs, such as wave function files with extension `.wfn`, must be generated by a run of BIGSTICK itself.

4.2 Defining the model space

A many-body model space is defined by a single-particle space, the valence Z and N , a total M value, a total parity (if applicable), and, optionally, truncations on that model space. Note that if you are carrying out what we call a secondary option, which starts from an existing wave function as stored in a `.wfn` file, BIGSTICK will automatically read from that file the information on the basis. Thus you only need to define the model space when carrying out a ‘primary’ option, that is, basically starting from scratch.

The single-particle space is defined one of two ways. Either you read in a file defining the single-particle space, or, for so-called *no-core shell model* calculations, automatically generate the basis in a pre-defined form.

For consistency, we generally refer to *orbits* as single-particle spaces labeled by angular momentum j but not j_z , while *states* refer to labels by both j and j_z .

Our default format for defining the single-particle space are derived from the format for OXBASH/NuShell/NuShellX files. A typical file is the `sd.sps` file:

```
! sd-shell
iso
3
      0.0  2.0  1.5  2
      0.0  2.0  2.5  2
      1.0  0.0  0.5  2
```

There is no particular formatting (spacing) to this file. Any header lines starting with `!` or `#` are skipped over. The first non-header line denotes about the isospin symmetry or lack thereof. `iso` denotes the single-particle space for both species is the same; one can still read in isospin breaking interactions. The second line (3 in the example above) is the number of single-particle orbits. The quantum

numbers for the single-particle orbits as listed are: n, l, j, w ; the first three numbers are real or integers, j is a real number. n is the radial quantum number, which play no role in **BIGSTICK** except to distinguish between different states. l is the orbit angular momentum and j is the total angular momentum; for the case of nucleons $j = l \pm 1/2$. In **BIGSTICK** the most important quantum number is j ; l is used internally only to derive the parity of each state.

While for most applications j is a half-integer, i.e., 0.5, 1.5, 2.5, etc., it can also be integer. In that case $l = j$ and one should interpret ‘protons’ and ‘neutrons’ as ‘spin-up’ and ‘spin-down.’ One can compute the electronic structure of isolated atoms, for example.

While n and l are not internally significant for **BIGSTICK**, they aid the human-readability of the `.sps` files; in addition, they can be invaluable as input to other code computing desired matrix elements.

BIGSTICK automatically unpacks each orbit to arrive at the $2j + 1$ single-particle states with different j_z .

The last ‘quantum number,’ w , is the *weight* factor, used for many-body truncations. We describe it later in Section 4.2.2. It must be a nonnegative integer.

BIGSTICK can handle any set of single-particle orbits; the only requirement is that each one have a unique set of n, l, j . (Although n and l are written above as real numbers, for historical reasons, they must have integer values. j can take either half-integer values or integer values with $l = j$; this latter we refer to as LS-coupling and is discussed in detail later on. All the j -values in a `.sps` file must be consistent, that is, all half-integer or all integer.)

So, for example, one could have a set of $l = 0, j = 1/2$ states:

```
iso
4
    0  0  0.5  0
    1  0  0.5  0
    2  0  0.5  0
    3  0  0.5  0
```

As of the current version of **BIGSTICK**, one cannot define completely independent proton and neutron spaces. One can however specify two variations. One can have `pns` instead of `iso`, which originally signalled to **BIGSTICK** to anticipate isospin-breaking interactions. This option is nearly obsolete. A more useful option is `wpn`, where protons and neutrons can have different weights:

```
wpn
3 3
    0.0  2.0  1.5  3
    0.0  2.0  2.5  2
    1.0  0.0  0.5  2
    0.0  2.0  1.5  3
    0.0  2.0  2.5  2
    1.0  0.0  0.5  3
```

While the proton and neutron orbits can have different weights, at this time the sets of quantum numbers must be the same and they must be listed in the same order. In the example above, we have proton $0d_{3/2}$, $0d_{5/2}$, and $1s_{1/2}$, and then the same for neutrons. Only the w values can be different.

The ordering of the single particle orbits is important and must be consistent with the input interaction files. If one uses our default-format interaction files, one must supply a `.sps` file.

It is possible to set environmental variables so that BIGSTICK automatically searches for `.sps` files in a different directory:

```
You can set a path to a standard repository of .sps/.sp files
by using the environmental variable BIG_SPS_DIR.
Just do :
export BIG_SPS_DIR = (directory name)
export BIG_SPS_DIR=/Users/myname/sps_repo
Currently BIG_SPS_DIR is not set
```

While we recommend the default format, we also allow for NuShell/NuShellX-compatible `.sp` files, which have a similar format. Like our default format, they also come in isospin-symmetric and proton-neutron format. An annotated example of the former is

```
! fp.sp
t           ! isospin-symmetric
40 20       ! A, Z of core
4           ! number of orbits
1 4         ! number of species, orbits per species
1 1 3 7     ! index, n, l, 2 x j
2 2 1 3
3 1 3 5
4 2 1 1
```

As with the default format, BIGSTICK will skip over any header lines starting with `!` or `#`. The next line, `t`, denotes an assumption of isospin symmetry. The next line, `40 20` are the A and Z of the core; these are not actually needed but are inherited.

The third non-header line, here `4` denotes the number of indexed orbits. The fourth non-header line, `1 4`, tells us there is just one ‘kind’ of particle with 4 orbits. The next four lines are the orbits themselves, with the orbital index, radial quantum number n , orbital angular momentum l , and twice the total angular momentum j . Here n distinguishes between different orbits which otherwise have the same l and j . In this example, n starts at 1, while in our other example n starts at 0. This makes no difference for BIGSTICK’s workings.

This can be contrasted with the `pn` option for the same space, which has separate indices for protons and neutrons.

```
! fppn.sp
pn
```

```

40 20
8
2 4 4
1 1 3 7
2 2 1 3
3 1 3 5
4 2 1 1
5 1 3 7
6 2 1 3
7 1 3 5
8 2 1 1

```

The main difference are in the third and fourth lines. There are a total of 8 orbits labeled, among two kinds or ‘species’ of particles, each with 4 orbits. The first 4 orbits are attributed to protons and the the next 4 to neutrons. While **BIGSTICK** accepts both formats, in practical terms it does not make a difference. At this time **BIGSTICK** does not all for fully independent proton and neutron spaces, and the ordering of proton and neutron orbits must be the same. (We hope to install the capability for more flexible spaces in the future.)

Notice that the **NuShell**-compatible **.sp** format does not include the weight-number w , which is assumed to be zero. Hence no many-body truncations are possible with these files.

If, instead, one uses an **MFDn**-formatted interaction file, one can use the **auto** option for defining the single-particle states.

```

Enter file with s.p. orbit information (.sps)
(Enter "auto" to autofill s.p. orbit info )
auto
Enter maximum principle quantum number N
(starting with 0s = 0, 0p = 1, 1s0d = 2, etc. )

```

The autofill option creates a set of single-particle orbits assuming a harmonic oscillator, in the following order: $0s_{1/2}, 0p_{1/2}, 0p_{3/2}, 1s_{1/2}, 0d_{3/2}, 0d_{5/2}$, etc., up to the maximal value N . It also associates a value w equal to the principal quantum number of that orbit, e.g., $2n + l$, so that N above is the maximal principal quantum number.

4.2.1 Particle-hole conjugation

BIGSTICK constructs the many-body basis states by listing the occupied particle states. Because the available single-particle space is finite, one can alternately list the unoccupied hole states. Such a representation can be advantageous if the single-particle space is more than half-filled, which only happens in phenomenological spaces: while the dimension of the Lanczos basis is unchanged, because of our jump technology the matrix elements can take much more space and memory. For example, consider diagonal matrix elements, $\langle \alpha | \hat{V} | \alpha \rangle$ which

are a sum over occupied states:

$$\langle \alpha | \hat{V} | \alpha \rangle = \sum_{a,b \in \alpha} V(ab, ab).$$

The number of terms in the sum is quadratic in the number of ‘particles’ in the system. Switching to holes can dramatically decrease the terms in this sum: if one has 12 single-particle states, for example, having two holes rather than ten particles makes a difference of a factor of 25! The overall scaling is not so simple, of course, for off-diagonal matrix elements (quickly: matrix elements of the form $\sum_b V(ab, cb)$, $a \neq c$, that is, between two states which differ only by one particle, go linearly in the number of particles, while those $V(ab, cd)$, $a \neq c, b \neq d$, that is, between two states which differ by two particles, are independent of the number of particles), in large model spaces one can see a big difference. In particular cases with a large excess of neutrons, so that we have a small number of protons but nearly fill the neutron space, can lead to enormous slow downs, as well as requiring many more jumps. Here transformation from particles to holes make for much greater efficiency. In order to obtain the same spectra and observables (density matrices), the matrix elements must be transformed via a *Pandya* transformation.

How to invoke particle-hole conjugation: When you are asked to enter the number of particles, you are told the maximum number of particles:

```
Enter # of valence protons (max 12 ), neutrons (max 12)
```

Simply enter the number of holes as a negative number, i.e.,

```
-2 -5
```

BIGSTICK will automatically carry out the Pandya transformation:

```
2  proton holes =          10  protons
5  neutron holes =          7  neutrons
```

You can conjugate protons, or neutrons, or both. If you enter the maximum number of particles in a space, BIGSTICK will automatically regard it as zero holes. Calculation of density matrices works correctly with particle-hole conjugation.

When written to file, hole numbers are also written as negative integers as a flag, and when post-processing, BIGSTICK will correctly interpret them.

We find there is little significant performance difference in spaces with up to about 20 single particle states, i.e. the *pf* shell, but beyond 20 it can become quite dramatic.

4.2.2 Truncation of the many-body space

Given a defined single-particle space, the basis states have fixed total M and fixed parity. If we allow all such states, we have a *full configuration* many-body

space. Sometimes, motivated either by physics or computational tractability, one wants to further truncate this many-body space. **BIGSTICK** allows a flexible scheme for truncating the many-body space which encompasses many, though not all, truncations schemes. We truncate the many-body space based upon single particle occupations. One could truncate based upon many-body quantum numbers, such as from non-Abelian groups, but that is beyond the scope of our algorithms.

Each single-particle orbit is assigned a weight factor w . This is read in from the `.sps` file or if the `autofill` option is used, is equal to the harmonic oscillator principal quantum number. w must be a nonnegative integer. If all orbits have the same w then no truncation is possible and **BIGSTICK** does not query about truncations.

w is treated as an additive quantum number: each basis state has a total W which is the sum of the individual w s of the occupied states. Because w is assigned to an orbit, it does not violate angular momentum or parity, and the total W is the same for all many-body basis states that are members of the same configuration, e.g., $0d_{5/2}^2(1s_{1/2})^1(0d_{3/2})^1$. Typically one assigns the same w to equivalent proton and neutron orbits (in principle one could assign different w s, which would break isospin, but we haven't explored this in depth).

Given the basis parameters, the single-particle orbits and their assigned w s and the number of protons and neutrons, **BIGSTICK** computes the minimum and maximum total W possible. The difference between these two is the maximal excitation:

```
Would you like to truncate ? (y/n)
y
Max excite =                20
Max excite you allow
```

The user chooses any integer between 0 and “Max excite.” **BIGSTICK** then creates all states with total W up to this excitation.

This scheme encompasses two major truncation schemes. The first kind of truncation is called a particle-hole truncation in nuclear physics, or sometimes n -particle, n -hole; in atomic physics (and occasionally in nuclear physics), one uses the notation ‘singles,’ ‘doubles,’ ‘triples,’ etc. To understand this truncation scheme, begin by considering a space of single-particle states, illustrated in Figure 4.1. Any single-particle space can be partitioned into four parts. In the first part, labeled ‘inert core,’ the states are all filled and remain filled. In the fourth and final part, labeled ‘excluded,’ no particles are allowed. Both the core and excluded parts of the single-particle space need not be considered explicitly, only implicitly. In some cases there is no core.

More important are the second and third sections, labeled ‘all valence’ and ‘limited valence’, respectively. The total number of particles in these combined sections is fixed at N_v , and this is the valence or active space.

The difference between the ‘limited valence’ and the ‘all valence’ spaces is that only some maximal number $N_l < N_v$ of particles are allowed in the ‘limited valence’ space. So, for example, suppose we have four valence particles, but only

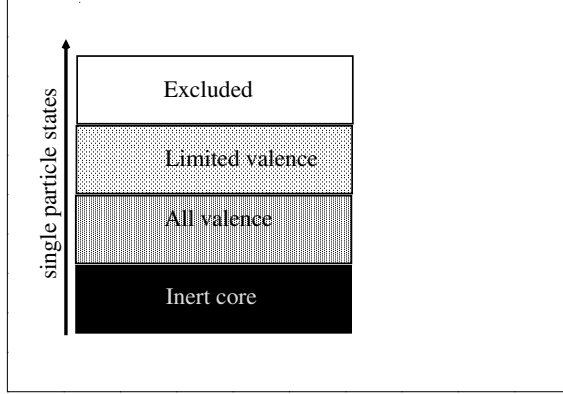


Figure 4.1: Segregation of single-particle space. 'Inert core' has all states filled. 'Excluded' disallows any occupied states. 'All valence' can have states up to the number of valence particles filled, while 'Limited valence' can only have fewer states filled (e.g. one, two, three...). See text for discussion. Figure taken from Johnson et al. [2013].

allow at most two particles into the 'limited valence' space. In this case the 'all valence' might contain four, three, or two particles, while the 'limited valence' space might have zero, one, or two particles. In more standard language, $N_l = 1$ is called 'one-particle, one-hole' or 'singles', while $N_l = 2$ is called 'two-particle, two-hole' or 'doubles', and so on. There are no other restrictions aside from global restrictions on quantum numbers such as parity and M .

The second truncation is commonly used in no-core shell model calculations, where center-of-mass considerations weigh heavily. For all but the lightest systems, one must work in the laboratory frame, that is, the wavefunction is a function of laboratory coordinates, $\Psi = \Psi(r_1, r_2, r_3, \dots)$. It is only the relative degrees of freedom that are relevant, however, so ideally one would like to be able to factorize this into relative and center-of-mass motion:

$$\Psi(r_1, r_2, r_3, \dots) = \Psi_{\text{rel}}(\vec{r}_1 - \vec{r}_2, \vec{r}_1 - \vec{r}_3, \dots) \times \Psi_{\text{CM}}(\vec{R}_{\text{CM}}) \quad (4.1)$$

(note that we have only sketched this factorization). In a harmonic oscillator basis and with a translationally invariant interaction, one can achieve this factorization exactly, *if* the many-body basis is truncated as follows ???:

- In the non-interacting harmonic oscillator, each single-particle state has an energy $e_i = \hbar\Omega(N_i + 3/2)$. Here N_i is the principal quantum number, which is 0 for the $0s$ shell, 1 for the $0p$ shell, 2 for the $1s-0d$ shell, and so on. The frequency Ω of the harmonic oscillator is a parameter but its numerical value plays no role in the basis truncation.
- We can then assign to each many-body state a non-interacting energy $E_{NI} = \sum_i e_i$, the sum of the individual non-interacting energies of each particle. There will be some minimum E_{min} and all subsequent non-interacting energies

will come in steps of $\hbar\Omega$ —in fact for states of the same parity, in steps of $2\hbar\Omega$.

- Now choose some N_{\max} , and allow only states with non-interacting energy $E_{NI} \leq E_{\min} + N_{\max}\hbar\Omega$. In practice, restricting states to the same parity means that the ‘normal’ parity will have $E_{NI} = E_{\min}, E_{\min} + 2\hbar\Omega, E_{\min} + 4\hbar\Omega, \dots, E_{\min} + N_{\max}\hbar\Omega$, while ‘abnormal’ parity will have $E_{NI} = E_{\min} + \hbar\Omega, E_{\min} + 3\hbar\Omega, \dots, E_{\min} + N_{\max}\hbar\Omega$.

This is sometimes called the N_{\max} truncation, the $N\hbar\Omega$ truncation, or simply the energy truncation. It is more complicated than the previous ‘particle-hole’ truncation. We identify with each principal quantum number N_i a major shell; for a $4\hbar\Omega$ we can excite four particles each up one shell, one particle up four shells, two particles each up two shells, one particle up one shell and another up three shells, and so on. While complicated, such a truncation allows us to guarantee the center-of-mass wavefunction is a simple Gaussian.

More generally, one can adjust the truncation scheme further, based upon skillful choice of single-particle w s. The assigned w s need not be contiguous; the only requirement is that they be nonnegative.

4.2.3 Advanced truncation options

All truncation is based upon the w weight factors. In most applications, both protons and neutron orbits have the same weights, and one typically truncates equally. A more general truncation scheme is possible.

First, as discussed in section 4.2, it is possible for proton and neutron orbits to have different values of w , if the `.sps` file has the ‘`wpn`’ format:

```
wpn
3 3
    0.0  2.0  1.5  3
    0.0  2.0  2.5  2
    1.0  0.0  0.5  2
    0.0  2.0  1.5  3
    0.0  2.0  2.5  2
    1.0  0.0  0.5  3
```

The dimensions of the proton and neutron orbits must be the same, as the order of all the quantum numbers besides w .

When asked for the truncation,

```
Would you like to truncate ? (y/n/?=more information)
```

choosing ‘p’ allows different truncation on protons and neutrons:

```
Max excite for sum, protons, neutrons?
(must be less than or equal to  8  4  4, respectively )
```

That is, the maximum values of $W_p + W_n$, W_p , and W_n , respectively. If you do not choose this option, then the limits are the same for all three.

4.2.4 How to handle ‘different’ proton-neutron spaces

As of the current version, **BIGSTICK** cannot handle different proton-neutron spaces. You can, however trick it into behaving that way, with a small cost. Both involve deft usage of the truncation and, in many cases, of particle-hole truncation.

Let’s consider two toy cases. First, suppose the proton and neutron spaces are entirely separate. For example, let’s suppose valence protons occupy only the $0f_{7/2}$ space and valence neutrons only the $1p_{3/2}$. The **.sps** file can look like:

```
iso
  2
0.0  3.0  3.5  0
1.0  1.0  1.5  1
```

By choosing a **Max excite** of zero, you will assure no particles are excited out of the $0f_{7/2}$ into the $1p_{3/2}$. (It is your responsibility to set up the correct interaction file. You do not have to include cross-shell matrix elements if they are not needed; however if they are included, they will induce an effective single-particle energy so choose wisely.)

A slightly more complex case: suppose you want protons active in $0f_{7/2}$, $1p_{3/2}$ and $1p_{3/2}$, and neutrons in $1p_{3/2}$, $1p_{3/2}$, and $0f_{5/2}$. Set up the **.sps** file

```
wpn
4  4
      0.0  3.0  3.5  0
      1.0  1.0  1.5  0
      1.0  1.0  0.5  0
      0.0  3.0  2.5  1
      0.0  3.0  3.5  0
      1.0  1.0  1.5  1
      1.0  1.0  0.5  1
      0.0  3.0  2.5  1
```

It is required that the proton and neutron orbits be the same, though the weight factors w is the last column can differ. Again, choosing **Max excite** of zero will keep the protons and neutrons in their respective valence spaces. If the valence spaces are significantly different, we strongly recommend utilizing particle-hole conjugation for the neutrons.

One can make the truncations even more complex, for example allow a few protons to be excited but no neutrons, by careful usage of the options provided.

4.3 Interaction files

After the model space is defined, **BIGSTICK** needs interaction matrix elements. All matrix elements are defined in the one-, two-, or possibly three-body-space. **BIGSTICK**’s job is to embed these matrix elements into a many-body space and

solve the eigenvalue problem. (Because three-body interaction files are highly specialized, we do not discuss their format.)

The default format for two-body interaction file is derived from `AXBASH/NuShell` and always ends in the extension `.int`. When entering the name of the file, only enter the name, not the extension, i.e., `usdb` not `usdb.int`; otherwise `BIGSTICK` will misinterpret the file.

```
! Brown-Richter USDB interaction
63      2.1117   -3.9257   -3.2079
  2  2  2  2    1  0  -1.3796
  2  2  2  1    1  0   3.4987
  2  2  1  1    1  0   1.6647
  2  2  1  3    1  0   0.0272
  2  2  3  3    1  0  -0.5344
  2  1  2  1    1  0  -6.0099
  2  1  1  1    1  0   0.1922
  2  1  1  3    1  0   1.6231
  2  1  3  3    1  0   2.0226
  1  1  1  1    1  0  -1.6582
  1  1  1  3    1  0  -0.8493
  1  1  3  3    1  0   0.1574
. . .
```

There is no specific spacing for this file. `BIGSTICK` will skip any header lines starting with `!` or `#`. The first line is

Ntbme *spe(1)* *spe(2)* *spe(3)* ...

where *Ntbme* is the number of *two-body matrix elements* (TBMEs) in the rest of the file, and *spe(i)* is the *single-particle energy* of the *i*th orbit. Only 10 single particle energies are on each line.

The rest of the file are the two-body matrix elements. This is defined as

$$V_{JT}(ab, cd) = \langle ab; JT | V | cd; JT \rangle, \quad (4.2)$$

where *a, b, c, d* label orbits, as ordered in the `.sps` file or as created by the autofill option; *J* and *T* are the total angular momentum and total isospin of the two-body states $|ab; JT\rangle$, which are normalized. This follows the convention of Brussaard and Glaudemans. Each matrix element is read in as

a *b* *c* *d* *J* *T* $V_{JT}(ab, cd)$

For input purposes, the order of *a, b, c, d* is not important (as long as one has the correct phase), nor is the ordering of the TBMEs themselves. When reading in the file, `BIGSTICK` automatically stores the matrix element according to internal protocols, appropriately taking care of any relevant phases.

Matrix elements that are zero can be left out, as long as *Ntbme* correctly gives the number of TBMEs in the file. More than one file can be read in; enter `end` to tell `BIGSTICK` you are finished reading in files.

4.3.1 Scaling and autoscaling

Empirical studies with phenomenological interactions have found best agreement with experiment if one scales the two-body matrix elements with mass number A . (There is some justification based upon the scaling of harmonic oscillator wave functions with A . A standard scaling factor is

$$\left(\frac{A_{\text{ref}}}{A}\right)^x \quad (4.3)$$

where A_{ref} is the reference mass number (typically A of the frozen core +2), A is the mass of the desired nucleus, and x is an exponent, typically around 1/3. To accomodate this scaling, when reading in the default format, BIGSTICK requests

```
Enter scaling for spes, A,B,X ( (A/B)^X ) for TBMEs
(If B or X = 0, then scale by A )
```

Typically the single particle energies are unscaled, but we allow for it. A typical entry, for example for the Brown-Richter USDA/B interactions, would be

```
1  18  24  0.3
```

Here the single particle energies are unscaled, the core has mass number 16, the target mass in this case has mass number 24, and the exponent is 0.3. Whoever provides the interaction has to provide the exponent. If unsure, just enter

```
1, 1, 1, 1
```

Many files used with NuShell have autoscaling. For example, for the USDA/B file, the first lines are

```
! 1=d3/2 2=d5/2 3=s1/2
! the first line has the three single-particle energies
! the - sign tells oxbash that the tbme have a mass dependence of the form
! [18/(16+n)]^0.3 where n is the number of valence particles
-63      1.9798   -3.9436   -3.0612   16.0000   18.0000 0.30000
```

A negative integer for the number of two-body matrix elements (here, -63) initiates autoscaling. The next three numbers are the single-particle energies, and the next numbers are A_{core} , the reference mass, and the exponent. If BIGSTICK encounters a negative integer for the number of two-body matrix elements, it autoscale the two-body matrix elements as described above. To turn off autoscaling, change -63 to 63.

4.3.2 Proton-neutron and other isospin-breaking formats

Often one needs to break isospin. There are three modifications of the default format which break isospin. In addition, *ab initio* inputs in the MFDn format, described in section 4.3.3, also generally break isospin.

The most robust format, which we recommend, is the explicit proton-neutron formalism. Here one has separate labels for proton and neutron orbits; however, at this time **the proton and neutron orbits must have the same quantum numbers and be listed in the same order**. For example, one might label the proton orbits $1 = 0d_{3/2}$, $2 = 0d_{5/2}$, and $3 = 1s_{1/2}$. Then the neutron orbits must be $4 = 0d_{3/2}$, $5 = 0d_{5/2}$, and $6 = 1s_{1/2}$.

While BIGSTICK generally allows for arbitrary order, for the proton-neutron matrix elements, the proton labels must be in the first and third columns, that is, for $V_J(ab, cd)$, a and c *must* be proton labels and b, d must be neutron labels.

The format for explicit proton-neutron input is very similar to the default format, but with twice as many orbits. Hence one must provide proton and neutron single particle energies. As an example, here is part of the file of the p -shell Cohen-Kurath matrix elements with good isospin:

```
! ORDER IS:  1 = 1P1/2    2 = 1P3/2
      15      2.419      1.129
      1  1  1  1      0  1      0.2440000
      1  1  1  1      1  0     -4.2921500
      2  1  1  1      1  0      1.2047000
      2  1  2  1      1  0     -6.5627000
      2  1  2  1      1  1      0.7344000
      2  1  2  1      2  0     -4.0579000
      2  1  2  1      2  1     -1.1443000
      2  2  1  1      0  1     -5.0526000
```

and here is an excerpt in proton-neutron formalism

```
      34      2.4190      1.1290      2.4190      1.1290
      1  3  1  3      0  1      0.24400
      1  1  1  1      0  1      0.24400
      3  3  3  3      0  1      0.24400
      1  3  1  3      1  1     -4.29215
      1  3  1  4      1  1     -0.85185
      1  3  2  3      1  1      0.85185
      1  3  2  4      0  1     -5.05260
      1  1  2  2      0  1     -5.05260
      3  3  4  4      0  1     -5.05260
      1  3  2  4      1  1      1.76980
      1  4  1  4      1  1     -2.91415
      1  2  1  2      1  1      0.73440
      3  4  3  4      1  1      0.73440
```

In no case are headers required, but they do help as a check for the definition of the orbits. BIGSTICK automatically checks that angular momentum and parity selections are not violated. In the explicit proton neutron format T is given in the sixth column but not actually used.

There is one more question of convention one must deal with: the normalization of the two-body states in the definition of matrix elements. All formats

assume two-proton and two-neutron states are normalized, and states with good isospin are normalized. Files set up for NuShellX, however, have *unnormalized* proton-neutron states.

BIGSTICK can read in default-format proton-neutron interactions with either normalized ('xpn' or explicit proton-neutron) or unnormalized ('upn' or unnormalized proton-neutron) conventions. In both cases the files also include proton-proton and neutron-neutron matrix elements, with normalized states.

The relationship between the two is

$$V_J^{xpn}(a_\pi b_\nu, c_\pi d_\nu) = \frac{2}{\sqrt{(1 + \delta_{ab})(1 + \delta_{cd})}} V_J^{upn}(a_\pi b_\nu, c_\pi d_\nu) \quad (4.4)$$

Here we have marked the orbits a, c as proton and b, d as neutron, but the Kronecker- δ s refer only to the quantum numbers n, l, j . For example, in the sd shell, with the labels mentioned above,

$$V_J^{xpn}(16, 25) = \sqrt{2} V_J^{upn}(16, 25)$$

because proton orbit 1 ($0d_{3/2}$) and neutron orbit 6 ($1s_{1/2}$) are different, but proton orbit 2 and neutron orbit 5 are both $0d_{5/2}$.

Unfortunately it is up to the user to know whether or not the file uses normalized or unnormalized proton-neutron states. If the file was originally produced for use with NuShellX, it is almost certainly the latter.

(This arises out of the conversion of normalized isospin wave function to normalized proton-neutron wave functions and the result matrix elements. One finds

$$V_J^{pn}(ab, cd) = \frac{\sqrt{(1 + \delta_{ab})(1 + \delta_{cd})}}{2} [V_{J,T=0}^{iso}(ab, cd) + V_{J,T=1}^{iso}(ab, cd)], \quad (4.5)$$

but the unnormalized convention yields the simpler

$$V_J^{upn}(ab, cd) = V_{J,T=0}^{iso}(ab, cd) + V_{J,T=1}^{iso}(ab, cd). \quad (4.6)$$

While our preference is for the former, given the prominence of the latter through NuShellX we include it as an option.)

In order to read in proton-neutron matrix elements in the default format, you must first tell BIGSTICK to expect it.

For xpn/upn formats, you MUST specify the format.

In this format proton and neutron orbits are sequential and do not overlap, E.g., proton orbits are 1,2,3 and neutron orbits are 4,5,6.

FOR NOW despite the distinct numbering the proton and neutron orbits must encompass the same space.

NOTE: upn format is typical for TBME files distributed with NuShell; xpn/upn files must have the name XXX.int, but enter XXX when requested.

As with default-format isospin-conserving files, the file name must be `xxxx.int`, but the user enters in just `'xxxx'`.

Also as with default-format isospin-conserving files, after entering the name of the file, the user is prompted for scaling. For maximal flexibility, there are two layers of possible scaling. The first is the standard phenomenological scaling:

```
Enter global scaling for spes, A,B,X ( (A/B)^X ) for TBMEs
(If B or X = 0, then scale by A )
1 18. 24. 0.3
```

These scalings are applied to all single particle energies and to all two-body matrix elements. In addition, one can enter in separate scaling factors for protons single-particle energies, neutron single-particle energies, proton-proton two-body matrix elements, neutron-neutron two-body matrix elements, and finally proton-neutron two-body matrix elements:

```
Enter individual scaling for: proton spes, neutron spes, pp TBMEs, nn TBMEs, p
n TBMEs
(If not sure, just enter 1 1 1 1 1 )
```

There are two alternate formats for isospin-breaking files which build upon the default format. These involve reading in separate files for proton-proton, neutron-neutron, and proton-neutron, or for isoscalar, isovector, and isotensor components. There are some tricky issues of definition, however. Thus we do not actively support these alternative formats, instead recommending the explicit proton-neutron format, whether normalized or unnormalized

One can mix all of these different formats. You can read in an isospin-conserving file, a proton-neutron format file, and so on, in any order. To stop reading in interaction files, enter `'end'` at the prompt.

4.3.3 MFDn format input

Another major configuration-interaction code is **MFDn** (Many-Fermion-Dynamics, nuclear version) out of Iowa State University [Sternberg et al., 2008]. While within **MFDn** there are several variations on conventions, we describe here the most common conventions.

Unlike the default format, to read in

The input file first line is

`nTBME (other stuff)`

where `nTBME` is the number of TBMEs in the file. Subsequent lines are

`a b c d J T Tre1 Hrel Vcoul V`

or

`a b c d J T Tre1 Hrel Vcoul Vpn Vpp Vnn`

Here all matrix elements are of the form $\langle ab; JT|V|cd; JT\rangle$, that is the matrix element between *normalized* two-body states with **a,b,c,d** labels of single particle orbits, **J** (and, optionally, **T**) are total angular momentum and isospin of the coupled two-body states.

Now for the matrix element. **Trel** is the relative kinetic energy, that is

$$\hat{T}_{\text{rel}} = \sum_{i<j} \frac{(\vec{p}_i - \vec{p}_j)^2}{2m}, \quad (4.7)$$

while

These matrix elements are computed in a harmonic oscillator basis for $\hbar\Omega = 1\text{MeV}$, and $A = 2$, and thus must be rescaled correctly for the A -body system.

Now to the final matrix elements. The actual Hamiltonian one wants is

$$\hat{H} = \hat{T}_{\text{rel}} + \hat{V}_{\text{rel}} + \hat{V}_{\text{ext}} + \beta_{c.m.}(\hat{H}_{\text{cm}} - \frac{3}{2}\hbar\Omega). \quad (4.8)$$

When you select an **MFDn**-format file, you will be prompted for the following:

```
For MFD-formatted input choose one of the following :
(I) No isospin breaking
(P) Explicit proton-neutron formalism
(C) Isospin breaking only through adding Coulomb
```

Almost always you should selection option ‘p’.

In order to use an *ab initio* file, you need to enter in the value of $\hbar\Omega$ for both the kinetic energy term and the center-of-mass Hamiltonian to push up spurious states:

```
Enter oscillator frequency (in MeV) and center-of-mass strength
```

You should know the frequency at which the file was created. The second term is $\beta_{c.m.}$. One adds $\beta_{c.m.}\hat{H}_{c.m.}$ to push up spurious states via the Palumbo-Lawson-Glocke method. Typical values are 1-10.

4.3.4 Three-body forces

While **BIGSTICK** has a validated capability for three-body forces, it is not optimized for large calculations; the main issue is storage of the large number of matrix elements. If you have the capability to generate three-body forces, please contact us, cjohnson@mail.sdsu.edu. We do not have the codes or capability to generate three-body forces for users.

If your distribution three-body forces are likely disabled. They can be re-enabled by setting the logical flag **threebodycheck = .true.** in module **flags3body** in the file **bmodules_3body.f90**. If this flag is enabled, **BIGSTICK** will query if you want to use three-body forces:

```
Do you want 3-body forces (y/n) ?
```


If you answer ‘n,’ BIGSTICK will proceed with just 2-body forces. If you answer ‘y,’ BIGSTICK will ask for the name of the file. Actually using three-body forces is complicated and beyond the scope of this current manual.

You can, however, used two-body forces but in three-body mode (the matrix elements are multiplied internally by $(\hat{N} - 2)/(A - 1)$ to turn them into genuine three-body forces), by answer ‘none’ to the question of the name of the file of three-body forces. Most users will not be interested in this.

4.4 Primary runtime options

Here we outline the major run time options, although some issues are discussed in more detail elsewhere. The main menu can be divided into two categories, primary and secondary. We discuss secondary options, which require results of a previous run, in section 4.6. In most primary runs one solves the matrix eigenvalue problem, which invokes the diagonalization options menu, discussed in section 4.5.

4.4.1 Autoinput

```
* (i) Input automatically read from "autoinput.bigstick" file      *
* (note: autoinput.bigstick file created with each nonauto run)    *
```

Each time BIGSTICK runs, it writes the user’s responses to a file `autoinput.bigstick`. This file can be edited or used as the basis of a batch file. The *autoinput* option, ‘i,’ will read in the `autoinput.bigstick` instead of taking responses from the user.

4.4.2 Standard or normal runs

```
* (n) Compute spectrum (default); (ns) to suppress eigenvector write up *
```

The *normal* run, ‘n,’ will generate the low-lying eigenspectrum and wave functions. This is the most common option. Two variations on it are **ns** which will compute the eigenspectrum and the J and T values, but not write the wavefunctions to a file, and **ne**, which will only compute eigenenergies. These latter options can save on time and file storage, but in most cases are not necessary.

4.4.3 One-body density matrices and occupations

One of the most important options for BIGSTICK is to generate the one-body density matrices, defined as

$$\rho_K^{fi}(a^\dagger b) = [K]^{-1} \langle J_f | (a^\dagger b)_K | J_i \rangle \quad (4.9)$$

where we use the choice of reduced matrix elements from Edmonds [1996],

$$\langle J_f | O_K | J_i \rangle = [J_f] (J_f M_f, K M | J_i M_i)^{-1} \langle J_f M_f | O_K M | J_i M_i \rangle \quad (4.10)$$

The advantage of this definition of the density matrix is that the reduced matrix element of a general, non-scalar one-body operator is just the density matrix \times the reduced matrix elements of that operator, that is

$$\langle \Psi_f, J_f | \hat{O}_K | \Psi_i, J_i \rangle = \sum_{ab} \langle a | \hat{O}_K | b \rangle \rho_K^{fi}(a^\dagger b) \quad (4.11)$$

where a, b are labels for single-particle orbits.

BIGSTICK has a number of options to generate density matrices.

```
* (d) Densities: Compute spectrum + all one-body densities *
* (dx[m]) Densities: Compute one-body densities from previous run (.wfn) *
*      optional m enables mathematica output *
```

The density matrix option ‘d’ runs just like the normal option, except at the end of the run it generates the one-body density matrices, which we describe more fully in section . If the interaction file has good isospin, then the one-body density matrices will be coupled up to good isospin. If the interaction file breaks isospin, the density matrices will be in proton-neutron format. If you use an interaction with good isospin but want the density matrices in proton-neutron format, use the option ‘dp.’

Two variations are the option ‘dx,’ which reads in a previously computed wave function file and from it computes the one-body density matrices, and ‘dxm,’ which does the same but generates the density matrices in a format readable by Mathematica. Technically these options are secondary.

```
* (p) Compute spectrum + single-particle occupations; (ps) to suppress wfn*
* (occ) single-particle occupations (from previous wfn) *
```

A restricted version of the one-body densities are the single-particle occupations. In principle given the former one can compute the latter, as described in section 5.1.2, but for convenience we give an option to do this directly. Option ‘p’ does this as in the normal option, but also writes the single-particle occupations to the .res file. Option ‘ps’ does the same but does not write the wave functions to a file on disk. Finally, the secondary option ‘occ’ reads in an existing wave function file and generates the single-particle occupations.

```
Single particle state quantum numbers
ORBIT :      1      2      3
  N :      0      0      1
  J :      3      5      1
  L :      2      2      0

State      E      Ex      J      T
  1    -62.78960    0.00000    2.500    1.500
  p occ:   0.136    1.590    0.275
  n_occ:   0.353    4.299    0.347
```

A detailed discussion of using one-body density matrices to get transition probabilities can be found in Chapter 5.1.

4.4.4 Other primary options

* (m) print information for Modeling parallel distribution *

The modeling option ‘m’ is useful for seeing if enough nodes and memory can be allocated for a large parallel run. We discuss it in more detail in section 8.1.3.

There are a couple of option most users are unlikely to use but which we mention nonetheless.

* (f) Self-consistent mean-field approximation (prepare pivot) *

* (c) Compute traces *

The ‘f’ or self-consistent mean-field option is an attempt to quickly get an good starting pivot for the ground state. Essentially what it does is replace the full Hamiltonian by a one-body approximation. On each iteration, **BIGSTICK** computes the one-body density matrices and by coupling the two-body force to those densities generates an effective, in general non-scalar one-body potential. This process is much faster than applying the full two-body interaction. It is inspired by but not the same as the Hartree-Fock approximation, which assumes a single Slater determinant.

The resulting approximate states are written to a **.wfn** file, which can be read back in as a pivot using the option ‘np.’ In our experiments this pivot speeds up convergence to the ground state but not excited states. More work especially on multishell systems have yet to be done. We have not fully tested this option in parallel.

BIGSTICK can also compute the trace of the Hamiltonian and the trace of the Hamiltonian squared, using the option ‘c.’ Specifically, it computes the centroid, which is the trace divided by the dimension, and the width, which is the square root of the variance.

```
for dimension    28503, centroid =    -43.544457, width =    11.298695
(saved to file trace.bigstick )
```

As shown above, the results are written to the file **trace.bigstick**. We caution users that in particular computing the width can be time consuming. We have not fully tested this option in parallel.

4.5 Diagonalization options

After the interactions files have been read in, **BIGSTICK** sets up the jump arrays for reconstructing the matrix elements on the fly. After that, the eigensolver menu comes up:

```
/ -----\
|
|   DIAGONALIZATION OPTIONS (choose one)   |
|
```

```

|
| (ex) Exact and full diagonalization (use for small dimensions only)
|
| (ld) Lanczos with default convergence (STANDARD)
| (lf) Lanczos with fixed (user-chosen) iterations
| (lc) Lanczos with user-defined convergence
|
| (td) Thick-restart Lanczos with default convergence
| (tf) Thick-restart Lanczos with fixed iterations
| (tc) Thick-restart Lanczos with user-defined convergence
| (tx) Thick-restart Lanczos targeting states near specified energy
|
| (sk) Skip Lanczos (only used for timing set up)
|
\ -----/

```

The full diagonalization option, ‘ex,’ creates the entire Hamiltonian matrix, stores it in memory, and solves it using the Householder algorithm as implemented in the LAPACK routine DSYEV. As such, it should not be used except for relatively small dimensions. On workstations one can solve up to $\sim 10^3$ in a few or tens of minutes. We have solved up to $\sim 10^4$, but that can take hours. In principle MPI versions of Householder exist, but we have not installed one, as one seldom has need for all eigensolutions of a very large matrix. (If you do not need wave functions or the angular momenta, choosing option ‘ne’ will speed this up dramatically, as DSYEV will run faster if one wants *only* eigenvalues.)

Under this option you can choose how many low-lying states, or all of them if you wish, to keep. These get written to file.

The primary eigensolver is the Lanczos algorithm, described in Chapter 7. Most of the time you will use option ‘ld’, the default Lanczos choice. Here you get asked

```

Enter nkeep, max # iterations for lanczos
(nkeep = # of states printed out )

```

BIGSTICK will run until the standard convergence criterion, see section 4.5.1, is satisfied, or unless the maximum number of iterations is exceeded. The latter must be specified to reserve memory for the Lanczos vectors. Although the Lanczos and related Arnoldi algorithms are among the most studied in applied mathematics, there is no simple, robust rule for the number of iterations needed. For phenomenological spaces, the ground state will often converge in under 50 iterations, the first 5 states in 100 to 150 iterations, and so on. For no-core shell model calculations, the time to convergence is usually longer.

The default convergence check is discussed in the next section, 4.5.1. If you want a fixed number of iterations without checking convergence, choose ‘lf.’ If you want finer control over convergence, choose ‘lc,’ discussed in 4.5.1.

The Lanczos vectors are stored in memory. For large-dimension cases, especially on a laptop or desktop, one can run out of memory just storing these vectors. Alternately, if one needs a large number of converged states, after a number of iterations reorthogonalization actually starts to take more time than matvec. A robust alternative is the thick-restart Lanczos algorithm [ref], which requires fewer vectors stored in memory but requires more iterations. While standard Lanczos finds the lowest N_{keep} eigensolutions with N_{iter} iterations, thick-restart has three numbers: $N_{\text{keep}} < N_{\text{thick}} < N_{\text{iter}}$. As described more fully in section 7.2, after N_{iter} iterations the approximate Hamiltonian is diagonalized, and N_{thick} of these eigenvectors are kept for restarting. This process is carried out until convergence.

```
td
  Enter # of states to keep, number of iterations before restarting
5 50
  Enter max # of restarts
10
```

As with standard Lanczos, the values chosen usually come with experience. We find we usually want $N_{\text{thick}} > \sim 3 \times N_{\text{keep}}$, and we take N_{iter} as large as practical. Specifically, BIGSTICK chooses

$$N_{\text{thick}} = \max(3N_{\text{keep}}, N_{\text{keep}} + 5),$$

as long as this is not larger than N_{iter} .

If you want a fixed number of iterations, choose ‘tf’, and you will be prompted for N_{keep} , N_{iter} , and then N_{thick} :

```
tf
  Enter # of states to keep, # of iterations before thick-restart
5 50
  Enter # of vectors to keep after thick-restart
  (Typical value would be          20 )
  ( Must be between          5   and          50 )
```

If you want to control the convergence, choose ‘tc’ and you will be prompted for convergence choices much as for standard Lanczos.

Finally, ‘tx’ is an experimental mode attempting to find highly excited states. It modifies thick-restart by choosing eigenpairs in the vicinity of a selected absolute energy. In our experience the convergence is not very good, but it does yield an eigenvector with very strong overlaps with the true eigenvectors in the vicinity of the target energy.

Option ‘sk’ is only for testing timing of set-up to this point.

4.5.1 Convergence

As BIGSTICK iterates, it checks for convergence. Every ten iterations it prints to screen the current N_{keep} lowest eigenvalues and the convergence criterion:

```

1      -135.86073
2      -133.92904
3      -131.25354
4      -131.02439
5      -129.53058

```

```

      80  iterations
(energy convergence  0.70356 > criterion  0.00100)

```

```

1      -135.86073
2      -133.92904
3      -131.25354
4      -131.02439
5      -129.53059

```

```

      90  iterations
(energy convergence  0.30353 > criterion  0.00100)

```

As far as we can tell from the literature, there is no robustly ideal convergence criterion for general Lanczos. Our default convergence is on energy: **BIGSTICK** takes the sum of the absolute value of the differences in energy between the current iteration and one previous, not only for the N_{keep} lowest energies but also for the next 5, and divides by the square root of $N_{\text{keep}} + 5$, that is,

$$\delta_{\text{conv}} \equiv \frac{\sum_{i=1}^{N_{\text{keep}}+5} |E_i^{\text{new}} - E_i^{\text{old}}|}{\sqrt{N_{\text{keep}} + 5}} \quad (4.12)$$

The reason for testing additional eigenvalues is to avoid the problem of plunging eigenvalues, well known to occur in Lanzos Whitehead et al. [1977]; it happens when by accident a low-lying state has a tiny overlap with the pivot or initial vector. We divided not by the number of energies compared but by the square root, because for a large number of energies one outlier could get washed out by many small deviations. For our purposes this has worked well enough, but it is not rigorously tuned.

If you want a different criterion, choose ‘lc’ on the diagonalization menu. One then gets a series of questions with which to tune the convergence, including comparing eigenvectors rather than eigenvalues:

```

Enter nkeep, max # iterations for lanczos
5 150
Enter how many ADDITIONAL states for convergence test
( Default= 5 ; you may choose 0 )
10

```

```

Enter one of the following choices for convergence control :
(0) Average difference in energies between one iteration and the last;
(1) Max difference in energies between one iteration and the last;

```

```

(2) Average difference in wavefunctions between one iteration and the last;
(3) Min difference in wavefunctions between one iteration and the last;
2
Enter desired tolerance
(default tol = 0.100E-04 )

```

Similar options are available for thick-restart Lanczos.

4.6 Secondary runtime options

Once BIGSTICK has generated wave function, it can go back and process further the wave functions. We discuss those options in detail here. Some of these were already mentioned in section 4.4. All of these options will ask for the name of a previously generated `.wfn` file.

```
Enter input name of .wfn file
```

You do *not* have to read in a `.sps` or similar file to define the model space; from the information in the `.wfn` file, BIGSTICK reconstructs the basis. Depending upon the option, you may be asked to enter names of appropriate files, such as interaction files.

We list these in the order they are presented in the menu, but the most important and commonly used option are ‘`x`,’ expectation value of a scalar operator, ‘`o`,’ apply a one-body non-scalar transition operator, and ‘`s`,’ the strength function option, all discussed below.

```
* (np) Compute spectrum starting from prior pivot *
```

Option ‘`np`’ allows you to choose a pivot, or initial starting vector, from a previously generated wave function. This might be useful, for example, if you wanted to try to get states of a particular quantum number such as J , although we do not currently have the capability to enforce this condition, and even if it is an exact quantum number numerical noise will allow states with other quantum numbers to creep in. Alternately, you can generate a guess of the ground state by the self-consistent mean-field option ‘`f`.’ Our experiments suggest this latter does lead to faster convergence of the ground state, but not excited states. A related and more widely useful option is the strength function option ‘`s`’ discussed below and in section .

```
* (dx[m]) Densities: Compute one-body densities from previous run (.wfn) *
* optional m enables mathematica output
```

Options ‘`dx`’ and ‘`dxm`’ read in a previously generated wave function and compute the one-body density matrices. The latter provides a Mathematica-friendly file format. More details about one-body density matrices are found in section .

* (occ) single-particle occupations (from previous wfn) *

This option computes the single-particle occupations from a previously generated wave function file.

4.6.1 Expectation value

* (x) eXpectation value of a scalar Hamiltonian (from previous wfn) *

The option 'x' allows you to compute the expectation value of an operator, which may have one-, two- (and in principle, three-) body components. It must be an angular momentum scalar and thus is treated as a Hamiltonian, and is read in exactly as Hamiltonians, with the scaling requests for scaling information. The results are written both to screen and, if named, the .res file"

STATE	E	J	T ²	<H >	(norm)
1	-92.7790	-0.0000	0.0000	499.287061	1.00000
2	-91.1196	2.0000	0.0000	488.826115	1.00000
3	-88.4779	2.0000	0.0000	509.922118	1.00000
4	-87.9781	4.0000	0.0000	452.853182	1.00000

4.6.2 Applying a one-body transition operator

One of BIGSTICK's important capabilities is to take a set of previously generated wave functions and apply a non-scalar one-body operator to them:

* (o) Apply a one-body (transition) operator to previous wfn and write out*
 * (oo) Apply a one-body (transition) operator with enforced orthogonality *
 * (that is, forced resulting pivot to be orthogonal to starting pivot *
 * to eliminate transition strength to original state) *

If you choose this option, you will be asked for the name of the input .wfn file as well as the name of an output .wfn file. Then you will be asked for a file with the reduced matrix elements of the operator, which must have the extension .opme:

Enter name of .opme file

Here is an annotated example .opme file:

```
! header: Gamow-Teller-like
iso                                ! assumes isospin
                                3      ! # of single particle orbits
1    0    2    1.5                ! index, n, l, j of orbits
2    0    2    2.5
3    1    0    0.5
                                1      ! J, T of transition
1    1    -2.68328                ! a, b  < a ||| 0  ||| b >
1    2    5.36656
```



```

2      1  -5.36656
2      2   5.01996
3      3   4.24264

```

The only formatting is the the first non-header line, here `iso`, must be flush against the left. The file must contain the single-particle orbits, and `BIGSTICK` checks against the orbits used to build the wave function. After the list of orbits, the J and T of the operator come, and then the the non-zero reduced matrix elements. Here, assuming isospin is a good quantum number, we have doubly-reduced matrix elements. Although there is a symmetry $\langle a||\hat{O}||b\rangle = (-1)^{j_a-j_b}\langle b||\hat{O}||a\rangle$, at this time `BIGSTICK` requires both elements.

Many transition operators do not preserve isospin. Therefore `BIGSTICK` can read in operators in an explicit proton-neutron symmetry:

```

! M1 matrix elements in the sd shell
pns
      3
1      0      2  1.5
2      0      2  2.5
3      1      0  0.5
      1      2
1      1      0.1568000      1.4481392      ! a b  proton m.e.    neutron m.e.
1      2      3.4710987      -2.8962784
2      1      -3.4710987      2.8962784
2      2      6.7871771      -2.7092204
3      3      3.3425579      -2.2897091

```

The code `pns` in the first non-header line signals that the matrix elements are in proton-neutron formalism, with the same list of orbital quantum numbers for protons and neutrons. The 2 in the 5th line also signals that one is breaking isospin. Thus in the list of reduced (in J only) matrix elements, the columns are for protons and neutrons, respectively.

`BIGSTICK` will read in all the wave functions $|\Psi_i\rangle$ from the initial wave function file, and write $\hat{O}|\Psi_i\rangle$ in the final wave function file. These wave functions will generally not be normalized and will not have good angular momentum or isospin. More on this elsewhere.

Currently, both the initial and final wave functions must be in the same basis. Thus, there are no explicit charge-changing transitions. To handle charge-changing transitions, one must use an interaction with good isospin and exploit isospin rotation, described in section 5.1.5. For transitions which change parity, one must use a basis with both parities, option 0 in the parity-selection menu.

4.6.3 Applying a two-body body scalar operator

* (a) Apply a scalar Hamiltonian to a previous wfn and write out *

Option ‘a’ works very similar to option ‘o:’ one reads in a previously-generated file of wave functions, applies an operator to each wave function, and writes the

results to another file. The difference is here the operator must be a one-plus-two-body *scalar* operator, that is, like a Hamiltonian. The files are read in the same as a Hamiltonian.

4.6.4 Generating strength function distributions

One of the most powerful and most useful capabilities is ‘s.’ the strength function distribution option. We give an overview here, with many more details of application in section 5.2.

Like all secondary options, the strength function option starts by prompting the user for a previously generated `.wfn` file. The user is then prompted for a Hamiltonian or Hamiltonian-like interaction file or files. Next the user must enter the number of iterations for Lanczos:

```
Fixed iterations ONLY:
Enter nkeep, # iterations for lanczos
(nkeep = # of states printed out )
```

The *only* way to make this option work is through standard Lanczos. The number of results to keep and the number of iteration depends upon the application; see section 5.2. Finally, the user must choose from the input file the pivot or starting vector, **a key decision**:

```
There are          5 wavefunctions
      5 states

STATE      E      J      <H >
  1      -92.7790   -0.0000    0.0000
  2      -91.1196    2.0000    0.0000
  3      -88.4779    2.0000    0.0000
  4      -87.9781    4.0000    0.0000
  5      -87.4348    3.0000    0.0000
Which do you want as pivot?
```

What happens next is that Lanczos runs normally, produces eigenvalues and eigenvectors and writes them to file. It also additionally computes the *overlap of the pivot with each of the eigenstates*, that is, $|\langle f | \text{pivot} \rangle|^2$:

Energy	Strength
17.30356	0.00007
49.98777	0.00123
110.94935	0.00956
184.33815	0.01945
249.75355	0.01641
301.54676	0.08014
383.34766	0.05833

```

428.29023    0.14090
498.95403    0.04282
534.87775    0.17398
588.87306    0.45712

```

```

-----

```

This is the *strength function* or strength distribution. If the starting vector has a norm different from one, this is noted

```

0.99999999895896363      = total input strength

```

and this is included in the strengths. The usefulness of this capability cannot be overestimated, and is discussed in depth in section 5.2.

4.6.5 Overlap or dot product of wave functions

```

* (v) Overlap of initial states with final states

```

*

The output eigenstates are written as vectors. Although most users are unlikely to need to use this, using the ‘v’ option BIGSTICK can compute the dot product between two such wave functions, including from different files. From the first file, you must choose a specific state:

```

Which do you want as initial state?

```

A second file is then opened (you can reopen the first file), and the initial state is dotted against each of them. The results are written to the file `overlap.dat`:

```

Initial state =          1
state      E      J      T      <i|f>      |<i|f>|^2
1      -87.10445 -0.0  0.0      0.99885      0.99771
2      -85.60214  2.0  0.0     -0.00000      0.00000
3      -82.98830  2.0  0.0      0.00000      0.00000
4      -82.73201  4.0  0.0      0.00000      0.00000
5      -82.03407  3.0  0.0     -0.00000      0.00000
6      -81.22187  4.0  0.0     -0.00000      0.00000
7      -79.76617 -0.0  0.0     -0.02326      0.00054

```

We found this option useful in validating other capabilities, such as the strength function capability.

4.7 Output files

BIGSTICK generates a number of output files. These fall into two broad categories. The most important output files have a name supplied by the user, e.g., `mg24` followed by an extension, e.g. `.res` or `.wfn`. Other files, which are not needed by most casual users, have the same standard name upon each run, ending in `.bigstick`.

Results. The most important file are the results files, which have an extension `.res`. When you initiate **BIGSTICK**, after the main menu choice, **BIGSTICK** almost always asks you for the name of the output files:

Enter output name (enter "none" if none)

If you enter “none” then several files are suppressed, in particular the results file.

The results file generally contains the output spectrum, e.g.

State	E	Ex	J	T
1	-41.39657	0.00000	0.000	0.000
2	-39.58581	1.81077	2.000	0.000
3	-37.08646	4.31012	4.000	0.000
4	-34.46430	6.93227	0.000	0.000
5	-33.56871	7.82786	2.000	-0.000

It also may contain one-body density matrices, or the results of strength function runs.

Wave functions. The `.wfn` file contains wave function information, stored in binary (or “unformatted.”) In addition to containing the wave function vectors, it has a header which contains enough information the basis can be recreated.

Autoinput. On each run **BIGSTICK** generates a file `autoinput.bigstick`. This saves the various input when run from the terminal. This is useful if one is making small tweaks to run, or to use as the basis for input directives. To use the autoinput file directly from terminal, choose ‘i’ at the opening menu.

Log file. The `.log` file summarizes information about the run, such as the date, time, **BIGSTICK** version number, dimensions, parallelization (number of MPI processes and OpenMP threads, for example) internal flag settings, and so on. While not needed by the casual user, they are useful to document the exact conditions under which a particular result ran and for debugging. If no output name is specified, this file is named `logfile.bigstick`.

4.7.1 Secondary files

BIGSTICK generates some intermediate files which are not needed for ordinary runs but in some cases can be useful. The most useful of these are the `.lcoef` files, which in an ordinary Lanczos run contains the Lanczos coefficients α_i, β_i . If no output name is specified, this file is called `lanczosvec.lcoef`.

There is also a file, `.lvec`, which originally contained the Lanczos vectors themselves. Usually empty, except in unusual cases in which the Lanczos vectors are stored on disk rather than in RAM.

4.7.2 Diagnosis files

BIGSTICK also generates a number of files which shed light upon its internal workings. These are primarily for development, tuning, and debugging.

`timingdata.bigstick` is the time spent in different matvec modes (SPE, PP, etc) on each MPI process.

`distrodata.bigstick` are the type and size of jumps stored on each MPI process.

4.8 Memory usage

The motivation for BIGSTICK's on-the-fly algorithm is to save memory over storing the nonzero many-body matrix elements. Despite this, BIGSTICK can still be quite memory-hungry. The main sinks of memory are: the Lanczos vectors themselves, the jumps factorizing the many-body matrix elements, and the uncoupled two-body matrix elements. Which dominates depends upon the system. For example, in large phenomenological calculations, the main memory usage is from Lanczos vectors. In no-core shell model calculations, it is typically the jumps, except for very light systems ($A = 3, 4$) where for large spaces the uncoupled matrix elements actually dominate.

BIGSTICK gives a report on memory usage. It also has some default caps on memory and will halt if these are violated. The default caps can be changed by the user.

Both in normal runs and in modeling runs, BIGSTICK produces a report:

RAM for 2 lanczos vector fragments (max)	:	3923.728 Mb
RAM for jumps in storage (total)	:	2353.914 Mb
Max RAM for local storage of jumps	:	177.069 Mb
RAM for uncoupled two-body matrix elements	:	0.017 Mb

The RAM report above is for the initial and final Lanczos vectors in matvec. In order to reorthogonalize, BIGSTICK also stores all Lanczos vectors. When run in MPI, these Lanczos vectors are distributed across many MPI processes:

```
Enter max number of Lanczos iterations
150
Assuming max memory per node to store Lanczos vectors    16.00000    Gb
Storage of Lanczos vectors distributed up across          128 nodes
Memory per node =    10.74658    Gb
```

The default memory caps can all be found in the module `flagger` in the file `bmodules_flags.f90`. The most important ones are

```
real    :: maxjumpmemory_default = 16.0    ! in Gb
real    :: maxlanczosstorage1 = 16.000    ! in Gb
```

These can be changed, though of course BIGSTICK must be recompiled.

Chapter 5

Applications

In this chapter we discuss in more detail applications of BIGSTICK, specifically one-body density matrices and one-body transition strengths.

5.1 One-body density matrices

BIGSTICK can be directed to compute the reduced one-body density matrices,

$$\rho_K^{fi}(ab) \equiv \frac{1}{\sqrt{(2K+1)}} \langle \Psi_f || [\hat{c}_a^\dagger \otimes \tilde{c}_b]_K || \Psi_i \rangle \quad (5.1)$$

where we use reduced matrix elements as defined in Appendix A. We use this particular definition because the reduced matrix element of a generic one-body operator is the sum of the density matrix elements \times the reduced matrix elements, namely,

$$\langle \Psi_f || \hat{O}_K || \Psi_i \rangle = \sum_{ab} \langle a || \hat{O}_K || b \rangle \rho_K^{fi}(ab). \quad (5.2)$$

It's important to note that $\langle a || \hat{O}_K || b \rangle$ are *matrix elements between single-particle states*, while the density matrices are matrix elements between *many-body states*. While some many-body codes compute the many-body matrix elements for specific operators, such as E2, M1, and so on, we chose for BIGSTICK to produce one-body density matrices, allowing the user to compute the transition matrix elements for *any* one-body operator.

For systems with good isospin one can also define “doubly-reduced” matrix elements, that is, reduced in both angular momentum and isospin;

$$\rho_{K,T}^{fi}(ab) \equiv \frac{1}{\sqrt{(2K+1)(2T+1)}} \langle \Psi_f || [\hat{c}_a^\dagger \otimes \tilde{c}_b]_{K,T} || \Psi_i \rangle \quad (5.3)$$

These are “doubly-reduced” matrix elements, that is, reduced in both angular momentum and isospin;

When you choose the density matrix option, BIGSTICK will write to the `.res` file (but not to screen) the density matrices, e.g.:

```

Initial state #    2 E = -85.60214 2xJ, 2xT =    4    0
Final state   #    1 E = -87.10445 2xJ, 2xT =    0    0
Jt =    2, Tt = 0          1
  1    1 -0.01957  0.00000
  1    2  0.18184  0.00000
  1    3  0.09721  0.00000
  2    1 -0.18184  0.00000
  2    2 -0.35744  0.00000
  2    3 -0.26323  0.00000
  3    1 -0.09721  0.00000
  3    2 -0.26323  0.00000

```

The first two lines are the labels and energies of the initial and final wave-functions; Jt and Tt are the angular momentum and isospin of the one-body operator, and, for example,

```

1    3  0.09721  0.00000

```

1 is the label of the first single-particle orbit, 3 the label of the second (as defined by the input `.sps` file), and the two real numbers are the $T = 0, 1$ one-body density matrix elements, that is,

$$\left\langle \Psi_2 \left| \left| \left[\hat{a}_1^\dagger \otimes \tilde{a}_3 \right]_{J=2, T=0} \right| \right| \Psi_1 \right\rangle = 0.09721$$

while that for $T = 1$ is, here, zero.

BIGSTICK actually has two options for densities. The option `d` will compute one-body densities with good isospin, where the output looks like (this example is ^{23}Ne in the *sd* shell with the USDB interaction):

```

Initial state #    1 E = -62.78960 2xJ, 2xT =    5    3
Final state   #    1 E = -62.78960 2xJ, 2xT =    5    3
Jt =    0, Tt = 0          1
  1    1  0.84730  0.28105
  2    2  8.32846  2.85633
  3    3  1.52286  0.13245

```

Alternately, there is the option `dp` which puts the density matrix elements into explicit proton-neutron form.

```

Initial state #    1 E = -62.78960 2xJ, 2xT =    5    3
Final state   #    1 E = -62.78960 2xJ, 2xT =    5    3
Jt =    0, proton      neutron
  1    1  0.16625  0.43288
  2    2  1.58968  4.29943
  3    3  0.47558  0.60124

```

5.1.1 Symmetries of density matrix elements

A useful symmetry relation is

$$\rho_{K,T}^{if}(ba) = (-1)^{j_a - j_b + J_i - J_f + T_i - T_f} \rho_{K,T}^{fi}(ab). \quad (5.4)$$

5.1.2 Particle occupations from densities

Particle occupations are the average number of particles in single-particle orbit for a given wave function. Although there is an option, ‘p,’ to compute this direction, you can also extract this information from the diagonal one-body density matrices. The total number of particles in orbit a is

$$n(a) = \frac{[j_a]}{[J_i]} \rho_{K=0}^{ii}(a^\dagger a) \quad (5.5)$$

If your densities are in proton-neutron format, you can extract the proton and neutron occupations separately. If you have your densities in isospin formalism, you can extract the total number of protons *and* neutrons in an orbit

$$n_\pi(a) + n_\nu(a) = \frac{[j_a][1/2]}{[J_i][T_i]} \rho_{K=0,T=0}^{ii}(a^\dagger a) \quad (5.6)$$

To separately extract proton and neutron occupation one must take careful account of the Clebsch-Gordan coefficients. One must have $f = i$, so that $J_f = J_i = J_0$ and $T_f = T_i = T_0$, as well as considering only $a = b$. Furthermore, the answer depends upon $T_z = (Z - N)/2$ (using the notation $[x] = \sqrt{2x+1}$)

$$n_\pi(a) = \frac{[\frac{1}{2}]}{[J_0]} \frac{[j_a]}{[T_0]} \frac{1}{2} \left(\rho_{K=0,T=0}^{ii}(a^\dagger a) + \frac{T_z \sqrt{3}}{\sqrt{T_0(T_0+1)}} \rho_{K,T=1}^{ii}(a^\dagger a) \right), \quad (5.7)$$

$$n_\nu(a) = \frac{[\frac{1}{2}]}{[J_0]} \frac{[j_a]}{[T_0]} \frac{1}{2} \left(\rho_{K=0,T=0}^{ii}(a^\dagger a) - \frac{T_z \sqrt{3}}{\sqrt{T_0(T_0+1)}} \rho_{K,T=1}^{ii}(a^\dagger a) \right). \quad (5.8)$$

5.1.3 Strengths from density matrix elements

Given some transition operator \hat{O}_K carrying definite angular momentum K , the transition strength between an initial and final state is just the square of the matrix element:

$$\left| \langle J_f M_f | \hat{O}_{KM} | J_i M_i \rangle \right|^2.$$

This is the matrix element that goes into Fermi’s golden rule for decay and transition rates.

But in most experimental situations we cannot pick out specific values of $M_{i,f}$ (unless we are doing an experiment with polarization). The final result must then *average* over initial states and *sum* over final states, that is,

$$\frac{1}{2J_i + 1} \sum_{M_i} \sum_{M_f} \left| \langle J_f M_f | \hat{O}_{KM} | J_i M_i \rangle \right|^2. \quad (5.9)$$

In most cases there is also implicitly a sum over M . (If not, the final result will be different.) Now we can use the Wigner-Eckhart theorem to rewrite the average/sum as:

$$\frac{1}{2J_i + 1} \sum_{M_f} \sum_{M_i} \sum_M |(J_i M_i, K M | J_f M_f)|^2 \left| (J_f || \hat{O}_K || J_i) \right|^2. \quad (5.10)$$

Now we can use the the selection rule $M_f = M_i + M$ to eliminate the sum over M_f and the orthogonality of the Clebsch-Gordan coefficients to sum over M_i and M

$$\sum_{M_i} \sum_M |(J_i M_i, K M | J_f M_i + M_f)|^2 = 1 \quad (5.11)$$

Thus we get the result in terms of reduced matrix elements,

$$\begin{aligned} \frac{1}{(2J_i + 1)} \sum_{M_i} \sum_{M_f} \sum_M \left| (J_f M_f | \hat{O}_{KM} | J_i M_i) \right|^2 \\ = \frac{1}{(2J_i + 1)} \left| (J_f || \hat{O}_K || J_i) \right|^2, \end{aligned} \quad (5.12)$$

As one often calls

$$\frac{1}{2J_i + 1} \left| (J_f || \hat{O}_K || J_i) \right|^2, \quad (5.13)$$

the *reduced* or *B-value*, written $B(\mathcal{O})$ (for example, $B(GT)$ for Gamow-Teller, $B(E2)$ for electric quadrupole, etc.), this says the strength for an operator is $B(\mathcal{O})$.

In the **BIGSTICK** code and most other shell-model codes, we actually compute transition strength using transition density matrix elements. The (doubly-reduced) density matrix between two states, Eq. (5.1), is defined so that the doubly reduced transition matrix element for a one-body operator $\hat{O}_{K,T}$ of angular momentum rank K and isospin rank T is

$$\langle \Psi_f ||| \hat{O}_{K,T} ||| \Psi_i \rangle = \sum_{ab} \rho_{K,T}^{fi}(ab) \langle a ||| \hat{O}_{K,T} ||| b \rangle, \quad (5.14)$$

where $\langle a ||| \hat{O}_{K,T} ||| b \rangle$ are the doubly-reduced one-body matrix elements of the operator. Although we provide some sample files, it is up to the user to generate them in general.

Although we use doubly-reduced matrix elements, the definition of B -values do *not* sum or average over ‘orientations’ in isospin space, because $T_z = (Z - N)/2$ is fixed. Hence we have to account for that by undoing the Wigner-Eckart reduction in isospin, so that, for non-charge changing transitions (e.g., γ -transitions),

$$\begin{aligned} B(\mathcal{O} : i \rightarrow f) &= \frac{1}{2J_i + 1} \left| (\Psi_f : J_f || \hat{O}_J || \Psi_i J_i) \right|^2 \\ &= \frac{1}{2J_i + 1} \left| (\Psi_f : J_f T_f ||| \hat{O}_{J,T} ||| \Psi_i J_i T_i) \right|^2 \times \frac{|(T_i T_z, T_0 | T_f T_z)|^2}{2T_f + 1}. \end{aligned} \quad (5.15)$$

Note the last line uses the result of Eq. (5.14).

5.1.4 Sample case: spin-flip

Let's consider a couple of simple cases, both in the sd shell with the Brown-Richter **usdb** (or 'universal sd -shell interaction, version b ') interaction. Let's consider the spin-flip operator $\vec{\sigma} = 2\vec{S}$, which has the following doubly-reduced matrix elements:

One-body matrix element	value
$\langle 0d_{3/2} \vec{\sigma} 0d_{3/2} \rangle$	-2.19089
$\langle 0d_{3/2} \vec{\sigma} 0d_{5/2} \rangle$	4.38178
$\langle 0d_{5/2} \vec{\sigma} 0d_{3/2} \rangle$	-4.38178
$\langle 0d_{5/2} \vec{\sigma} 0d_{5/2} \rangle$	4.09878
$\langle 1s_{1/2} \vec{\sigma} 1s_{1/2} \rangle$	3.46410

The nuclide ^{19}F , which has only one valence proton and two valence neutrons, has, with appropriate scaling of the matrix elements, the low-lying spectrum:

State	E	Ex	J	T
1	-23.86096	0.00000	0.500	0.500
2	-23.78367	0.07729	2.500	0.500
3	-22.09059	1.77037	1.500	0.500
4	-21.26237	2.59858	4.500	0.500
5	-19.25724	4.60371	6.500	0.500

The density matrix from the second state ($J = 5/2$) to the third ($J = 3/2$) state is, up to some overall phases,

Initial state #	2	E =	-23.78367	2xJ,	2xT =	5	1
Final state #	3	E =	-22.09059	2xJ,	2xT =	3	1
Jt =	1,	Tt =	0				1
1	1	-0.08640	-0.01635				
1	2	0.44978	-0.36112				
1	3	0.01255	-0.09014				
2	1	-0.16826	0.08815				
2	2	-0.00280	-0.35352				
3	1	-0.03483	0.07521				
3	3	0.28978	-0.20874				

Because the Pauli matrix $\vec{\sigma}$ carries one unit of angular momentum and no isospin, we only use the ($Tt=0$) set of matrix element (upper left column). **BIGSTICK** also generates the transition matrix elements for $Jt = 2, 3$, and 4, not shown. Applying (5.15), we get a $B(\sigma : 2 \rightarrow 3) = 1.2609$

A second case is ^{20}Ne . The ground state is at -40.4723 MeV, which the first $J = 1, T = 0$ state, state #25, is at -27.8364 MeV (or 12.636 MeV excitation energy). The density matrix is

Initial state #	1	E =	-40.47233	2xJ,	2xT =	0	0
Final state #	25	E =	-27.83635	2xJ,	2xT =	2	0
Jt =	1,	Tt =	0				1
1	1	0.00069	0.00000				

1	2	0.14575	0.00000
1	3	-0.10567	0.00000
2	1	0.18722	0.00000
2	2	-0.04822	0.00000
3	1	-0.02309	0.00000
3	3	0.28308	0.00000

and $B(\sigma : 1 \rightarrow 25) = 0.3597$.

5.1.5 Charge-changing transitions

Charge-changing transition such as Gamow-Teller are a little more subtle. If we have isospin-conserving interactions, so that our initial and final states have good isospin, we can use *isospin rotation* so that we don't have to change basis. If we want to have a transition

$$^A_Z X_N \rightarrow ^A_{Z\pm 1} Y_{N\mp 1},$$

that is, from some initial $T_{z,i} = (Z - N)/2$ to some final $T_{z,f} = (Z - N)/2 \pm 1$, we must work in the basis with the smaller T_z ; then both initial and final states will be somewhere in the spectrum. What we want to calculate is

$$\left| \langle \Psi_f : J_f, T_f T_{z,f} | \hat{\mathcal{O}} | \Psi_i : J_i, T_i T_{z,i} \rangle \right|^2,$$

but what we can actually calculate with BIGSTICK is

$$\left| \langle \Psi_f : J_f, T_f T_z | \hat{\mathcal{O}} | \Psi_i : J_i, T_i T_z \rangle \right|^2.$$

Fortunately this can be accomplished with only a small modification of the above procedure:

$$\begin{aligned} B(\mathcal{O} : i \rightarrow f) &= \frac{1}{2J_i + 1} \left| \langle \Psi_f : J_f | \hat{\mathcal{O}}_J | \Psi_i J_i \rangle \right|^2 \\ &= \frac{1}{2J_i + 1} \left| \langle \Psi_f : J_f T_f | \hat{\mathcal{O}}_{J,T} | \Psi_i J_i T_i \rangle \right|^2 \times \frac{|(T_i T_{z,i}, T \pm 1 | T_f T_{z,f})|^2}{2T_f + 1}, \end{aligned} \quad (5.16)$$

where the difference between Eq. (5.15) and (5.16) is in the isospin Clebsch-Gordan. There is, however, one more subtle point in treating the isospin raising/lowering operator, τ_{\pm} . If one treats τ as a rank-1 spherical tensor in isospin space, one can show that

$$\tau_{\pm} = \frac{1}{\sqrt{2}} \tau_{1,\pm 1}. \quad (5.17)$$

Therefore, formally, in the above calculations, we are actually using $2^{-1/2} \tau_{1,0}$ in our calculation, and then rotating to a charge-changing transition.

It's always good to have a way to check calculations, and in the case of Gamow-Teller it's the Ikeda sum rule, which says

$$\sum_f B(\vec{\sigma} \tau_+ : i \rightarrow f) - B(\vec{\sigma} \tau_- : i \rightarrow f) = 3(N - Z) \quad (5.18)$$

independent of the initial state i . Here the isospin raising operator τ_+ changes a neutron into a proton and hence is the operator for β^- decay, while the isospin lowering operator τ_- changes a proton into a neutron and hence is the operator for β_+ decay. This assume our convention that protons are isospin ‘up’ and neutrons isospin ‘down;’ many authors have opposite conventions.

5.1.6 Sample case: ^{19}F

Let’s calculate the Gamow-Teller B-value. The matrix elements for Gamow-Teller are the same as for $\vec{\sigma}$ as shown above except multiplied by $\sqrt{3/2}$. Then using the $T_t = 1$ one-body density matrix elements, we get $B(\text{GT}: 2 \rightarrow 3) = 1.3990$.

For ^{20}Ne , there is at $J = 1, T = 1$ state at -29.3066 MeV (or 11.166 MeV excitation energy, state #15); the density matrix is

Initial state #	1	E =	-40.47233	2xJ, 2xT =	0	0
Final state #	15	E =	-29.30659	2xJ, 2xT =	2	2
Jt =	1, Tt = 0		1			
1	1	0.00000	0.05163			
1	2	0.00000	0.09951			
1	3	0.00000	-0.03397			
2	1	0.00000	0.18236			
2	2	0.00000	0.32717			
3	1	0.00000	-0.03311			
3	3	0.00000	-0.08363			

Here $B(\text{GT}) = 0.1654$, for either β_+ or β_- .

5.2 Strength function option

One important capability of BIGSTICK is using the Lanczos algorithm to efficiently compute transition strength function distributions and to decompose a wavefunction using a scalar operator.

5.2.1 Decomposition

We’ll start with decomposition of a wavefunction using a scalar operator, because operationally it is the most straightforward. Suppose you have a wavefunction, $|\Psi\rangle$, which you have previously computed using BIGSTICK and have stored in a `.wfn` file; further suppose you have some operator \hat{O} which is an angular momentum scalar, which in turn means its matrix elements can be stored in a file just like a Hamiltonian. This operator \hat{O} in turn has eigenpairs,

$$\hat{O}|\Phi_\omega\rangle = \omega|\Phi_\omega\rangle. \quad (5.19)$$

We can always expand $|\Psi\rangle$ into the eigenstates of \hat{O} :

$$|\Psi\rangle = \sum_{\omega} c_{\omega} |\Phi_{\omega}\rangle \quad (5.20)$$

and the fraction of the wavefunction $|\Psi\rangle$ labeled by ω is simply

$$|\langle\Phi_\omega|\Psi\rangle|^2 = |c_\omega|^2.$$

This is particularly useful when \hat{O} is the Casimir of some group or subgroup, such as total orbital angular momentum \hat{L}^2 or total spin \hat{S}^2 . In that case we say we *decompose* the wavefunction $|\Psi\rangle$ into its L - or S - components.

BIGSTICK can carry out this decomposition easily. What you need is, first, a previously computed wavefunction in some `.wfn` file, and a file or files which contain the matrix elements of the decomposing operator.

To do this:

1. From the initial menu choose the option ‘s’:

```
* (s) Strength function (using starting pivot )          *
:
Enter choice
s
```

Note: the *pivot* is the starting vector; here it is the wavefunction you wish to decompose. BIGSTICK can only decompose one wavefunction at a time.

2. Enter name of file containing the wavefunction to be decomposed (i.e., the pivot):

```
Compute strength function distribution using previous wfn
Enter input name of .wfn file
mg24
```

Here the choice of the wavefunction file is `mg24.wfn`; you do not include the extension. At this point, BIGSTICK reads in some information from the `.wfn` file:

```
testing magic number      31415926      31415926
dimbasischeck=            28503
Valence Z, N =             4             4
Single particle space :
N      L      2xJ
      0      2      3
      0      2      5
      1      0      1

Total # of orbits =          3
2 x Jz =              0
```

The ‘magic number’ is a test of internal consistency to make sure, first, BIGSTICK is correctly reading the file (in particular if the binary file was created

on a different platform) and also between different versions of **BIGSTICK** if the information protocol has changed.

From this information **BIGSTICK** reconstructs the basis and checks the dimensions agree. It then asks for the output file:

Enter output name (enter "none" if none)

After this, **BIGSTICK** will make the standard inquiries for the Hamiltonian. When decomposing a wavefunction, the ‘Hamiltonian’ is actually an angular momentum scalar which is a Casimir of the group or sub-group; for example, it could be \hat{S}^2 or \hat{L}^2 . Such files are in the same format as any interaction file. While we provide a sample operator, it is up to the user to generate these files.

After the interaction file(s) have been read in, you must enter in the number of iterations and number of states to keep (**BIGSTICK** automatically chooses a fixed-iteration run for Lanczos):

Enter nkeep, # iterations for lanczos
(nkeep = # of states printed out)

Exactly how many iterations to choose requires some knowledge of the group, or, specifically, knowledge of the eigenvalues of the Casimir, and, in many cases, a few trials. Remember that the irreps of the group are labeled by the eigenvalues of the Casimir, which means the eigenvalues are highly degenerate. The number of iterations needed should be no greater than the number of distinct eigenvalues. So, for example, if one has 8 nucleons and is decomposing via spin, the values of S can be 0, 1, 2, 3, or 4. Therefore the number of iterations should be no more than 4 (because one wants a total dimension of 5). Often one can use fewer iterations. If you use too many iterations, you will get duplication of eigenvalues or, worse, unconverged duplicate eigenvalues.

Finally, **BIGSTICK** will print out a list of the starting states in the pivot file, and their energies and J and T values, and ask you to choose a pivot:

```

There are          5 wavefunctions
          5 states

STATE      E          J          <H >
  1      -92.7790     -0.0000      0.0000
  2      -91.1196      2.0000      0.0000
  3      -88.4779      2.0000      0.0000
  4      -87.9781      4.0000      0.0000
  5      -87.4348      3.0000      0.0000
Which do you want as pivot?
```

Hence if you want to decompose the $J = 3$ state, enter 5.

Immediately after reading in the pivot, **BIGSTICK** will print out the norm of the input pivot (that generally does play a role in this kind of decomposition, but will in transition strength functions):

```
0.999999998379788          = total input strength
```

After carrying out the specified Lanczos iterations, the result will look something like this, depending on how many iterations:

Energy	Strength
0.00000	0.63545
2.00000	0.33880
6.00000	0.02515
12.00000	0.00059
20.00000	0.00000

The ‘energies’ on the left are the eigenvalues of the operator you are using to decompose the wavefunction, here \hat{S}^2 . Hence the $J = 3$ state (or state 5 in the above example), is 63.5% $S = 0$, 33.9% $S = 1$, and so on. These results are written to the standard `.res` file.

5.2.2 Transition strength function distributions: the basics

Often we want the transition function between two states, that is $|\langle \Psi_f | \hat{O} | \Psi_i \rangle|^2$ where \hat{O} is some one-body transition operator, for example the $E2$ or $M1$ transition operator. (As always, we assume the reader is familiar with these concepts.) If one only wants one or two transitions, one can compute those using the one-body density matrices, which we describe above in 5.1.3.

But sometimes we want many transitions to many final (or ‘daughter’) states from a single initial (‘parent’) state, for example if we want to profile ‘giant’ resonances. We can do this using BIGSTICK in three to four steps. The first step is to generate and write to file an initial wavefunction.

The second step is to apply a one-body operator, \hat{O} . The matrix elements of the one-body operator must be stored in file with extension `.opme`, with the format defined in the next section. To apply a one-body operator, choose option ‘o’ at the opening menu:

```
* (o) Apply a one-body (transition) operator to previous wfn....
```

BIGSTICK will then ask for the name of the input `.wfn` file and an output name, required here. After reconstructing the basis from the information in the input `.wfn` file, it will ask:

```
Enter name of .opme file
```

The matrix elements of the one-body operator are read in from a file with extension `.opme` (‘operator matrix element’). While we distribute some sample `.opme` files with BIGSTICK, in general it is up to the user to generate such files. The format of such files are

```

iso                                ! indicating good isospin
                                3      ! # of single-particle orbits
1    0    2    1.5                ! index of orbit, n, l, and j
2    0    2    2.5
3    1    0    0.5
                                1    0      ! J and T of operator
1    1   -2.19089 ! a, b < a ||| 0 ||| b >
1    2    4.38178
2    1   -4.38178
2    2    4.09878
3    3    3.46410

```

BIGSTICK first checks the list and order of single-particle orbits agrees with that of the read-in wavefunction. BIGSTICK will then read in the matrix elements of the one-body operator and apply it to *each* wavefunction stored in the input `.wfn` file and write them to a new output `.wfn` file.

The final step is to run BIGSTICK again, this time with the strength function option ‘s,’ using the wavefunction generated in the second step as input. This time, when BIGSTICK asks for the interaction file name, you should use the *same* file(s) to generate the initial state, because you are diagonalizing the Hamiltonian.

As with decomposition, BIGSTICK will now carry out a fixed number of iterations and print out the transition strength. Because it includes the norm of the input pivot, these strengths can be greater than 1.

An important question is that of convergence. As you probably know, in the Lanczos algorithm the extremal eigenpairs converge first, with interior eigenpairs converging later. This is true as well for the strengths described above: the extremal strengths will converge quickly to strengths (and eigenenergies) of extremal levels, but interior strengths will often not be converged; instead they will be some sort of ‘local average’ of strengths.

What looks like a bug is actually a feature. In practice one often doesn’t need each and every strength to be fully converged. Instead we only need integrals over the strengths to be converged, and this does happen. While we can only refer the reader to the literature [REFS], we can state that the *moments* of the distributions of strengths do converge. In fact, if one carries out N Lanczos iterations, one has $\sim 2N$ moments of the distribution correctly. Hence often only thirty or fifty iterations suffice.

As an example, consider ^{20}Ne in the *sd* shell with the `usdb` interaction. If we apply the σ operator, with the matrix elements given above, and then apply the strength function option `s`, the output will look something like:

```

0.38353481218057317      = total strength
      35 iterations

Energy   Strength
-----
-40.47233  0.00000

```


-38.72564	0.00000
-36.29706	0.00000
-33.77148	0.00000
-32.88892	0.00000
-30.18655	0.00000
-27.83635	0.11991
-25.73419	0.00003
-25.49045	0.17314
-24.73655	0.01326
-22.86589	0.00027
-22.24522	0.01542

Notices the strength at -27.836 MeV (which is the $J; T = 1; 0$ state) is 0.11991; using the Clesbsch-Gordan coefficients gives a factor of 3, or a total strength of 0.3597, which agrees with our previous result.

There is a small bug in this lovely ointment: it assumes we have treated angular momentum (and isospin) correctly, a topic we now turn to.

5.2.3 Transition strength functions with good angular momentum

In the prior subsection we glided over questions of angular momentum, which we treat more carefully here. An important question is correct calculation of the B -values, as defined in Eq. (5.13) above, which assume an average over final states and a sum over final states. But what we computed in the previous section was

$$|\langle \Psi_f | \hat{\mathcal{O}} | \Psi_i \rangle|^2$$

where the states have fixed M and fixed T_z , because, as currently written, both initial and final wavefunctions must be in the same basis. (We plan at a later date to write a separate tool which will allow one to apply an operator from a wavefunction in one basis to a wavefunction in a different basis.) If you want the applied operator to change parity, then both parities must be included in the basis (option 0 when entering parity in the initial calculation of wavefunctions).

To extract the B -value, one has to invoke the Wigner-Eckart theorem:

$$B(\mathcal{O} : i \rightarrow f) = \frac{1}{2J_i + 1} \left| \langle \Psi_f : J_f || \hat{\mathcal{O}}_J || \Psi_i J_i \rangle \right|^2 = \frac{2J_f + 1}{(2J_i + 1)} \left| \frac{\langle \Psi_f : J_f M | \hat{\mathcal{O}}_{J0} | \Psi_i : J_i M \rangle}{(J_i M, J0 | J_f M)} \right|^2. \quad (5.21)$$

If one has $J_i = 0$ (which can only happen if $M = 0$), then the B -value is straightforward to calculate:

$$B(\mathcal{O} : i \rightarrow f) = (2J + 1) |\langle \Psi_f : J0 | \hat{\mathcal{O}}_{J0} | \Psi_i : 00 \rangle|^2.$$

It's more complicated with $J_i \neq 0$; in that case the triangle rule, $|J_i - J| \leq J_f \leq J_i + J$ is in effect, and in fact the state produced by **BIGSTICK**,

$$\hat{O}_K |\Psi_i : J_i\rangle$$

will be an admixture of states of different J_f . Thus one needs an additional step, of projecting out states of good angular momentum *after* applying the transition operator but *before* carrying out the strength function option via Lanczos. Fortunately we already know how to do this via decomposition as discussed in 5.2.1.

Therefore, to properly carry out calculation of strength functions, you will need (a) files for the interaction, (b) a file for the one-body transition operator, and (c) files with matrix elements of \hat{J}^2 and, separately, T^2 (you only need the latter if your transition operator has isospin rank 1). Then carry out the following steps:

(1) With your interaction use option (n) or similar option, generate a **.wfn** file containing an initial state;

(2) Use option (o) to apply the one-body operator (note this will be applied to *every* wavefunction in the file);

(3) If your initial state has $J_i \neq 0$, you will need to filter out a state of good J_f for every possible J_f ; in fact what you will do will be to decompose the state from step (2) into its components J_f . Here you use option (s). If there are N possible values of J_f you only need to do $N - 1$ iterations.

If your transition operator has $T = 1$ and your initial $T_i \neq 0$, you will need to further decompose into possible T_f states.

(4) Finally, use option (s) again, but this time with the original interaction, to get the strength function distribution. You will have to apply the Wigner-Eckart theorem as in Eq. (5.21), but now that step (3) guarantees a definite value of J_f (and, if needed, T_f), you can carry this out.

You will have to repeat for each possible final value of J_f .

Here is an example using ^{19}F : if we choose the $J = 5/2$ state (state #2) as the pivot and apply $\vec{\sigma}$, and then use the strength function option with \hat{J}^2 ,

```
1.3726179231928042      = total strength
      3 iterations
```

Energy	Strength
3.75000	1.13861
8.75000	0.05630
15.75000	0.17771

This means 83% of the pivot has $J = 3/2$ and only 4.08% has $J = 5/2$. Next we run the strength function again on the second state using the original (usdb) interaction:

```

1.0000000433422094      = total strength
      35 iterations

Energy   Strength
-----
-23.86096  0.00000
-23.78367  0.00000
-22.09059  0.66440
-21.26237  0.00000
-19.25723  0.00000

```

Note that the wavefunction is normalized when it is read in. (By the way, the zeroes show up because although there is no strength to them, or very little, round-off error allows them to grow during Lanczos.) We have to multiply $0.66440 \times 1.13861 = 0.756$ to get the ‘raw’ strength, which here is $|\langle J_f M | \vec{\sigma} | J_i M \rangle|^2$, then we have to follow Eq. (5.21):

$$0.756 \times \frac{2(3/2) + 1}{2(5/2) + 1} \times \frac{1}{|(5/2 \ 1/2, 1 \ 0 | 3/2 \ 1/2)|^2} = 0.756 \times \frac{4}{6} \times \frac{1}{2/5} = 1.260$$

which agrees with our previous result! Note that you have to do each step with care; if don’t scale the two-body matrix elements, you will get different results.

5.2.4 Gamow-Teller with strength function option

Charge-changing transitions such as Gamow-Teller are a straightforward generalization but require even more care. Here one transitions from a state with some initial $T_{z,i}$ to some final $T_{z,f} = T_{z,i} \pm 1$. Because, as of the time of this writing, BIGSTICK requires the same initial and final basis, we have to choose $T_{z,0} = \min(\text{abs}(T_{z,i}), \text{abs}(T_{z,f}))$ and invoke isospin rotation. Typically you will have to filter both J and T . The B-value is given by

$$\begin{aligned}
B(\mathcal{O} : i \rightarrow f) &= \frac{1}{2J_i + 1} \left| \langle \Psi_f : J_f, T_f T_{z,f} | \hat{\mathcal{O}}_J | \Psi_i : J_i, T_i T_{z,i} \rangle \right|^2 = \\
&\frac{2J_f + 1}{(2J_i + 1)} \left| \frac{\langle \Psi_f : J_f M, T_f T_{z,0} | \hat{\mathcal{O}}_{10,10} | \Psi_i : J_i M, T_i T_{z,0} \rangle}{(J_i M, J_0 | J_f M)} \right|^2 \\
&\times \left| \frac{(T_i T_{z,i}, 1 \pm 1 | T_f T_{z,f})}{(T_i T_{z,0}, 1 \ 0 | T_f T_{z,0})} \right|^2
\end{aligned} \tag{5.22}$$

where the last line uses the isospin Wigner-Eckart theorem to transform from the isospin frame the calculation is carried out in, to the physically desired isospin frame.

We use BIGSTICK's strength function option **s** to compute the matrix element $|\langle \Psi_f : J_f M, T_f T_{z,0} | \hat{O}_{10,10} | \Psi_i : J_i M, T_i T_{z,0} \rangle|^2$. This is slightly involved. We give now a detailed example, again with ^{19}F . After applying the Gamow-Teller operator, we filter out first with \hat{J}^2 , using the second state as the pivot:

```
0.72792934307990387      = total strength
      3  iterations
```

Energy	Strength
3.75000	0.61870
8.75000	0.01998
15.75000	0.08925

and then we filter this with \hat{T}^2 (applied to the first state, that is, the one with $J = 3/2$)

```
0.99999997896552772      = total strength
      3  iterations
```

Energy	Strength
0.75000	0.77481
1.10978	0.00000
3.75000	0.22519

and then finally applying the strength function with the **usdb** interaction:

```
1.00000000877719881      = total strength
     35  iterations
```

Energy	Strength
-23.86096	0.00000
-23.78367	0.00000
-22.09059	0.87875
-21.26237	0.00000
-19.25722	0.00000

Thus in this example the 'raw' transition strength is that for the $J; T = 5/2; 1/2 \rightarrow 3/2; 1/2$ which is

$$0.61870 \times 0.77481 \times 0.87875 = 0.42125.$$

This now has to be converted to a Gamow-Teller B-value by Eq. (5.22):

$$B(GT) = \frac{2 \cdot 3/2 + 1}{2 \cdot 5/2 + 1} \times \frac{0.42125}{|(5/2 \ 1/2, 1 \ 0 | 3/2 \ 1/2)|^2} \times \left| \frac{(1/2 \ -1/2, 1 \ 1 | 1/2 \ 1/2)}{(1/2 \ -1/2, 1 \ 1 | 1/2 \ 1/2)} \right|^2$$

A good exercise is to compute low-lying transitions two ways, first with density matrices, and then via the strength function option, to confirm they agree with each other. For Gamow-Teller, one can and should verify results by using the Ikeda sum rule.

One can also use other operators for projection, for example, using center-of-mass to project out nonspurious states in no-core shell model calculations.

We plan to later allow two-body transition operators, but as of version 7.8.1 these options have not yet been installed.

Chapter 6

A peek behind the curtain

Although this manual discusses in detail how to use BIGSTICK, it only outlines the algorithms and actual coding. The distribution includes an *Inside Guide* which, while incomplete, contains many more details on the code, and the source code itself is heavily commented. Even so, it is a complex code, with more than seventy Fortran files and on the order of 70,000 lines. Several of the Fortran files are for specialized applications most users will not care about, as well as features slated for obsolescence.

There are ways to get BIGSTICK to present more information about its inner workings, as well as ways to exert more control over the algorithm. A number of logical flags turn behaviors on and off. The most important flags, and some default settings, are found in `bmodules_flags.f90`. Some additional flags for output are in the module `io` in `bmodules_main.f90`, and other flags can be found elsewhere. To detail all the possibilities would expand this already long manual by a significant amount.

In this chapter we outline the major steps BIGSTICK takes in carrying out a ‘normal’ run, as well as telling the curious user how to print out an explicit representation of the basis and of the many-body Hamiltonian matrix.

6.1 A normal run

Here are the steps BIGSTICK carries out in a ‘normal’ run, that is, setting up a many-body Hamiltonian and finding the low-lying extremal eigensolutions.

- BIGSTICK sets up the basis.
- BIGSTICK counts up the number of jumps (data needed for constructing the Hamiltonian on-the-fly).
- BIGSTICK gathers the interaction data.
- If running in parallel, BIGSTICK computes the distribution

- BIGSTICK generates the data needed (on a specific MPI process if running in parallel), specifically the jumps and the decoupled matrix elements.
- BIGSTICK sets up storage for the Lanczos vectors.
- BIGSTICK begins Lanczos iterations.
- Upon completion of Lanczos, BIGSTICK constructs the low-lying eigenvectors. It resets the jumps and computes angular momentum and isospin as expectation values. Eigenvalues and eigenvectors are written to file. If density matrices requested, BIGSTICK resets jumps for one-body operators and computes.
- Upon finishing, BIGSTICK reports on timing and closes down.

6.2 Explicit representation of the basis

Through factorization and other tricks, BIGSTICK only implicitly stores the basis and the Hamiltonian. In actual operations, BIGSTICK stores information on pieces of Slater determinants, which we call “haikus.” Haikus are organized by quantum numbers, as are the action of single-fermion creation and annihilation operators on the haikus. These latter we called “hops” and from them we construct jumps, and from jumps we construct many-body matrix elements, and so on. Once the hops are created the haikus are not needed, and once the jumps are constructed the hops are not needed.

It can be useful, however, to have explicit representations of both the basis and of the many-body Hamiltonian. In standard runs, the final eigenvectors are written to file with extension `.wfn`. These files are unformatted to save space. Furthermore the detailed basis information is not saved; instead any basis BIGSTICK is constructed in a standard order, and when reading a `.wfn` file BIGSTICK swiftly reconstructs the basis.

From the main menu, however, the option ‘`t`’ will write out both the basis and the eigenvectors in explicit, human-readable form, to a file with extension `.trwfn` (originally written as an input to Petr Navratil’s density code TRDENS).

Here is an annotated example output from the p -shell, using the Cohen-Kurath interaction. First is a header describing the nucleus:

```

      4      ! valence Z
      4      ! valence N
ckpot ! name of INTERACTION FILE
19.45492      ! HW (approx)      12
      1      ! # of majors shells
12      1 ! total p+n s.p.s, # shells core
      0      ! Nmax (excitations)
           51 ! # of many-body configurations = basis dimension
      1      ! parity, +
      0      ! 2 x Jz
```

5 ! # of eigenstates kept

Next is a list of the eigenenergies, and numerical J and T values; the latter are written as real numbers. Reading across we have E , J , and T .

-71.04467	3.6173283E-06	-4.1723251E-07
-66.39703	2.000000	-3.8743019E-07
-58.59552	1.000001	-4.4703484E-07
-57.57795	3.6235542E-06	-4.1723251E-07
-57.54143	4.000000	-2.9802322E-07

Then comes a list of the single particle state quantum numbers. Reading across we have label, n (number of radial nodes), l , $2 \times j$, $2 \times j_z$, and $2 \times t_z$:

1	0	1	1	-1	1
2	0	1	3	-1	1
3	0	1	3	-3	1
4	0	1	1	1	1
5	0	1	3	1	1
6	0	1	3	3	1
7	0	1	1	-1	-1
8	0	1	3	-1	-1
9	0	1	3	-3	-1
10	0	1	1	1	-1
11	0	1	3	1	-1
12	0	1	3	3	-1

So single-particle state 1 is a $0p_{1/2}$ with $j_z = -1/2$ and is a proton, single-particle state 2 is $0p_{3/2}$ with $j_z = -1/2$ and is a proton, etc.

Finally we have a listing of the 51 many-body basis states and their amplitudes for the first five eigenstates:

1	2	3	5	8	10	11	12
0.1989746094	0.1647298932	-0.0000000709	-0.0788139924	-0.1036236882			
1	2	3	5	7	10	11	12
0.0805789307	0.0938614532	0.0855044648	-0.1246829554	-0.0399925224			
1	2	3	4	8	10	11	12
-0.0805789307	-0.0938614309	0.0855044946	0.1246829703	0.0399925113			
1	2	3	4	7	10	11	12
-0.0582427122	-0.0433882587	0.0000000068	0.0952372476	0.0186970048			
1	2	3	6	7	8	11	12
0.0825415403	-0.0342168659	-0.0919694155	-0.0390651748	0.1794791222			
1	2	3	6	7	8	10	12
0.0697834268	-0.0333072022	0.0155492499	-0.1079809889	0.1385308802			
1	2	3	6	9	10	11	12
-0.1513192952	0.0582505427	-0.0000000329	0.0562667921	-0.3108672500			
2	3	4	5	7	8	11	12
0.0560085103	-0.0361775197	-0.0000000029	-0.0111581217	0.1036224812			
...							

So the first many-body basis states has occupied single particle states 1,2,3,5 (protons) and 8,10, 11, and 12 (neutrons). The five real numbers following are the amplitudes for this basis state for the five eigenstates whose eigenenergies are given above. You can see an example of factorization: the basis states 1 and 2 have the same proton occupancies loop over neutron occupancies; while basis states 3 and 4 have the same proton occupancies (but different from basis states 1 and 2) and loops over the *same* neutron occupancies as basis states 1 and 2. By adding up the j_z values, the proton “Slater determinants” have $M_p = -2$ and the neutron Slater determinants have $M_n = +2$. The next basis states, number 5 through 7, have $M_p = -M_n = -1$. In this way **BIGSTICK** builds up the basis. As shown in the example, in constructing the basis via factorization, the innermost loop is over neutron Slater determinants while the outer is over protons.

We recommend against using this option on a regular basis, because writing this information to a file is slow, and **BIGSTICK** does not have postprocessing options for this format. Nonetheless it can be useful for understanding what is going on, and could be a basis for a user’s own post-processing code.

6.3 Explicit representation of the Hamiltonian

The many-body Hamiltonian can also be explicitly generated. If from the Lanczos menu you choose ‘**ex**’ for exact or full diagonalization, the entire Hamiltonian is created and explicitly stored and solve using the LAPACK routine **DSYEV**. This occurs in the routine **exactdiag_p** in the file **blanczos_main.f90**. If the basis dimension is less than 100, **BIGSTICK** will automatically write out the Hamiltonian matrix elements to a file **ham.dat**. This occurs around line 1246; the user can edit this part of the code to control the dimension cutoff for writing out (in general we do not encourage writing out for large dimensions) as well as the format.

Chapter 7

Lanczos algorithm

Today the most common way to find all the eigenpairs of a Hermitian (here, real and symmetric) matrix is first reduce the matrix to tridiagonal form via a sequence of unitary transformations, the Householder algorithm, and then solve the resulting tridiagonal matrix via QL decomposition with implicit shifts (or so we've been told). But for the very large dimensions of standard CI calculations, one neither can extract all eigenpairs nor does one want to: beyond a certain point one expects intruder many-body states to enter and eventually dominate, rendering our calculated states lacking physical meaning. (For the novice: an intruder state is a state outside the designated model space. There is no simple way, at least for the non-expert, to determine *where* we expect intruders to dominate.)

Instead we turn to the Lanczos and related algorithms. Lanczos is part of a family of so-called Arnoldi algorithms, which iteratively construct a new orthonormal basis, the Krylov subspace. In this new basis the Hamiltonian is tridiagonal, but unlike the Householder algorithm, one does not need to fully carry out the transformation. The Lanczos algorithm is simple, beautiful, and powerful, though like all algorithms it is not without its own limitations.

7.1 Standard Lanczos algorithm

The Lanczos algorithm is exceedingly straightforward. We will summarize it here, though we will not explicate it in detail. Starting from some initial vector $|v_1\rangle$, called the *pivot*, one iteratively generates a sequence of orthonormal vectors

$\{|v_i\rangle, i = 1, k\}, \langle v_i|v_j\rangle = \delta_{ij}$:

$$\begin{aligned}
\hat{H}|v_1\rangle &= \alpha_1|v_1\rangle + \beta_1|v_2\rangle \\
\hat{H}|v_2\rangle &= \beta_1|v_1\rangle + \alpha_2|v_2\rangle + \beta_2|v_3\rangle \\
\hat{H}|v_3\rangle &= \beta_2|v_2\rangle + \alpha_3|v_3\rangle + \beta_3|v_4\rangle \\
&\dots \\
\hat{H}|v_i\rangle &= \beta_{i-1}|v_{i-1}\rangle + \alpha_i|v_i\rangle + \beta_i|v_{i+1}\rangle \\
&\dots \\
\hat{H}|v_k\rangle &= \beta_{k-1}|v_{k-1}\rangle + \alpha_k|v_k\rangle
\end{aligned} \tag{7.1}$$

Each iteration generates a new Lanczos vector. If we stop at the $k - 1$ th iteration, we have k Lanczos vectors and a k -dimension Krylov subspace. Using orthonormality of the vectors, one can show that in this basis, the Hamiltonian is tridiagonal:

$$H_{i,i} = \langle v_i|\hat{H}|v_i\rangle = \alpha_i, \tag{7.2}$$

$$H_{i,i+1} = H_{i+1,i} = \langle v_i|\hat{H}|v_{i+1}\rangle = \beta_i. \tag{7.3}$$

and all other matrix elements are zero.

The specific steps for creating the next Lanczos vectors are straightforward:

- (1) $|w_i\rangle = \hat{H}|v_i\rangle$ *Initial matvec on vector i;*
- (2) $\alpha_i = \langle v_i|w_i\rangle$ *dot product to get α_i ;*
- (3) $|w_i\rangle \leftarrow |w_i\rangle - \alpha_i|v_i\rangle$ *orthogonalize against initial vector i;*
- (4) *If $i > 1$ $|w_i\rangle \leftarrow |w_i\rangle - \beta_{i-1}|v_{i-1}\rangle$ orthogonalize against prior vector i-1;*
- (5) $\beta_i = \sqrt{\langle w_i|w_i\rangle}$ *find norm to get β_i ;*
- (6) $|v_{i+1}\rangle = \beta_i^{-1}|w_i\rangle$ *Normalize to get i+1th Lanczos vector.*

If we had perfect arithmetic, this would be sufficient: the new Lanczos vector $|v_{i+1}\rangle$ would be guaranteed to be orthogonal to all previous vectors. But we don't have perfect arithmetic, and due to round-off noise, small components of prior Lanczos vectors will creep in and eventually grow exponentially.

This requires us to enforce orthogonality against all prior Lanczos vectors:

- (4)(alt.) *For $j = 1$ to $i-1$: $|w_i\rangle \leftarrow |w_i\rangle - |v_j\rangle\langle v_j|w_i\rangle$*

If one does not reorthogonalize, eventually one gets 'ghost eigenvalues', or repetitions of the same eigenvalues. It is this need for reorthogonalization that keeps Lanczos from supplanting Householder as the go-to algorithm for full tridiagonalization of Hermitian matrices.

There has been much discussion and experimentation around partial reorthogonalization, but there is no one clearly successful recipes. **BIGSTICK** fully reorthogonalizes against all prior vectors; in most cases (a few hundred iterations) reorthogonalization work does not overwhelm matvec work.

You might notice that if one extended the *for* loop in our alternate step (4), we would already get step (3). Because of finite arithmetic, order matters. We find better results if we first compute α_i and then orthogonalize against all other vectors, rather than as a last step.

It is possible for a user to experiment with these fine tweaks in **BIGSTICK**. The Lanczos iterations are found in subroutine `lanczos_p` in file `blanczoslib1.f90`.

One can estimate the workload from reorthogonalization. Again, let N be the dimension of the vector space. Each projection requires a dot product and a subtraction, or about $2N$ operations. For k Lanczos vectors, one has $k - 1$ iterations. For the j th iteration one orthogonalizes against j vectors or $2Nj$ operations; thus for $k - 1$ iterations one has $2N$

Obviously with full reorthogonalization full Lanczos transformation to tridiagonal form becomes expensive; hence the dominance of the Householder algorithm for complete diagonalization.

7.2 Thick-restart Lanczos

Sometimes there is insufficient storage for the number of Lanczos vectors required for convergence. An alternative is the *thick-restart Lanczos*. In standard Lanczos there are essentially two dimensions, N_{keep} , the number of converged states desired, and N_{iter} , the number of iterations (typically k above). But one must store $N_{\text{iter}} + 1$ Lanczos vector, which can be prohibitive. For thick-restart Lanczos, there is an additional dimension N_{thick} , with $N_{\text{keep}} < N_{\text{thick}} < N_{\text{iter}}$, and is the dimension of the Krylov subspace when restarting. In other words, one iterative creates a subspace of dimension $N_{\text{iter}} + 1$, but then truncates down to dimension N_{thick} , and then adds additional vectors back up to a subspace of dimension $N_{\text{iter}} + 1$, truncate back down, and repeat until convergence. The advantage is that N_{iter} is much smaller than would be needed for standard Lanczos.

Thick-restart Lanczos follows this basic outline:

1. Start with some initial Lanczos pivot vector $|v_1\rangle$ as usual.
2. Carry out k Lanczos iterations so that you have $k + 1$ Lanczos vectors $|v_i\rangle, i = 1, k + 1$, and a truncated $k + 1 \times k + 1$ Hamiltonian matrix \mathbf{T}^{k+1} .
3. Diagonalize the $k \times k$ submatrix \mathbf{T}^k .
4. From the eigenpairs of step (3), choose the N_{thick} lowest states. These will form the “new” Lanczos eigenvectors. In addition, keep $|v_{k+1}\rangle$ and use this as the restarting vector for Lanczos.
5. Now restart Lanczos, but instead of starting with $|v_1\rangle$, start with $|v_{k+1}\rangle$ which is our new $|v_{N_{\text{thick}}+1}\rangle$.
6. Iterate until you have again $k + 1$ Lanczos vectors and an truncated $k + 1 \times k + 1$ Hamiltonian matrix \mathbf{T}^{k+1} . This new matrix will no longer be tridiagonal, but it will have a simple form, given below.

Now let’s describe this in more detail. Suppose we have carried out k Lanczos iterations, so that we have a total of $k + 1$ vectors $|v_i\rangle$, including the pivot. Then

the transformed Hamiltonian, which is the Hamiltonian in the basis $\{|v_i\rangle\}$, looks like

$$\mathbf{T}^{k+1} = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & 0 & \dots & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & \dots & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & \dots & 0 \\ \vdots & & & & & \\ 0 & 0 & \dots & \alpha_{k-1} & \beta_{k-1} & 0 \\ 0 & 0 & \dots & \beta_{k-1} & \alpha_k & \beta_k \\ 0 & 0 & \dots & 0 & \beta_k & \alpha_{k+1} \end{pmatrix} \quad (7.4)$$

Suppose, however, we only diagonalize \mathbf{T}^k , that is, stopping at the k th column and row, with \mathbf{L} being the $k \times k$ unitary matrix of eigenvectors, that is,

$$\sum_{j=1}^k (\mathbf{T}^k)_{ij} L_{j\mu} = L_{i\mu} \tilde{E}_\mu, \quad (7.5)$$

for $\mu = 1, k$. Here \tilde{E}_μ are the approximate eigenenergies. If we apply the unitary transform \mathbf{L} to the first k vectors, that is, introducing

$$|v'_\mu\rangle = \sum_{i=1}^k |v_i\rangle L_{i,\mu}, \mu = 1, k \quad (7.6)$$

and $|v'_{k+1}\rangle = |v_{k+1}\rangle$ the transformed matrix, which is the Hamiltonian the basis $\{|v'_i\rangle\}$, now becomes

$$\begin{pmatrix} \tilde{E}_1 & 0 & 0 & 0 & \dots & \beta_k L_{k1} \\ 0 & \tilde{E}_2 & 0 & 0 & \dots & \beta_k L_{k2} \\ 0 & 0 & \tilde{E}_3 & 0 & \dots & \beta_k L_{k3} \\ \vdots & & & & & \\ 0 & 0 & \dots & \tilde{E}_{k-1} & 0 & \beta_k L_{k,k-1} \\ 0 & 0 & \dots & 0 & \tilde{E}_k & \beta_k L_{kk} \\ \beta_k L_{k1} & \beta_k L_{k2} & \dots & \beta_k L_{k,k-1} & \beta_k L_{kk} & \alpha_{k+1} \end{pmatrix} \quad (7.7)$$

The key to thick-restart Lanczos is to judiciously truncate this. If you want, as is the usual case, to get the lowest N_{keep} states, truncate to some N_{thick} (with $N_{\text{keep}} < N_{\text{thick}} < k$) vectors, that is, to

$$|v'_1\rangle, |v'_2\rangle, |v'_3\rangle, \dots |v'_{N_{\text{thick}}}\rangle$$

plus the last Lanczos vector, $|v_{k+1}\rangle$, then the truncated Hamiltonian looks like

$$\begin{pmatrix} \tilde{E}_1 & 0 & 0 & 0 & \dots & \beta_k L_{k1} \\ 0 & \tilde{E}_2 & 0 & 0 & \dots & \beta_k L_{k2} \\ 0 & 0 & \tilde{E}_3 & 0 & \dots & \beta_k L_{k3} \\ \vdots & & & & & \\ 0 & 0 & \dots & \tilde{E}_{N_{\text{thick}}-1} & 0 & \beta_k L_{kN_{\text{thick}}-1} \\ 0 & 0 & \dots & 0 & \tilde{E}_{N_{\text{thick}}} & \beta_k L_{kN_{\text{thick}}} \\ \beta_k L_{k1} & \beta_k L_{k2} & \dots & \beta_k L_{k,N_{\text{thick}}-1} & \beta_k L_{kN_{\text{thick}}} & \alpha_{k+1} \end{pmatrix} \quad (7.8)$$

Now declare $|v_{k+1}\rangle$ to be the new $|v'_{N_{\text{thick}}+1}\rangle$ and start the Lanczos iterations on it:

$$H|v'_{N_{\text{thick}}+1}\rangle = \quad (7.9)$$

$$\beta_k L_{k1}|v'_1\rangle + \beta_k L_{k2}|v'_2\rangle + \dots + \alpha_{N_{\text{thick}}+1}|v'_{N_{\text{thick}}+1}\rangle + \beta_{N_{\text{thick}}+1}|v'_{N_{\text{thick}}+2}\rangle$$

This first step is not a tridiagonal relation; furthermore, although our new $|v'_{N_{\text{thick}}+1}\rangle$ is the same as our old $|v_{k+1}\rangle$, and our new $\alpha_{N_{\text{thick}}+1}$ is the same as the old α_{k+1} , the new vector $|v'_{N_{\text{thick}}+2}\rangle$ is *not* the same as $|v_{k+1}\rangle$ would have been had we continued the previous iteration, although the former contains the latter as a component, because the iteration step is different.

Now one continues iterations $N_{\text{thick}}+2, N_{\text{thick}}+3, \dots, k+1$. Then one diagonalizes the approximate T^k again, although it is no longer a pure tridiagonal, and in fact looks like:

$$\begin{pmatrix} \tilde{E}_1 & 0 & 0 & 0 & \dots & \beta_k L_{k1} & 0 & \dots \\ 0 & \tilde{E}_2 & 0 & 0 & \dots & \beta_k L_{k2} & 0 & \dots \\ 0 & 0 & \tilde{E}_3 & 0 & \dots & \beta_k L_{k3} & 0 & \dots \\ & \vdots & & \ddots & & & & \\ 0 & 0 & \dots & \tilde{E}_{N_{\text{thick}}-1} & 0 & \beta_k L_{kN_{\text{thick}}-1} & 0 & \dots \\ 0 & 0 & \dots & 0 & \tilde{E}_{N_{\text{thick}}} & \beta_k L_{kN_{\text{thick}}} & 0 & \dots \\ \beta_k L_{k1} & \beta_k L_{k2} & \dots & \beta_k L_{kN_{\text{thick}}-1} & \beta_k L_{kN_{\text{thick}}} & \alpha_{N_{\text{thick}}+1} & \beta_{N_{\text{thick}}+1} & \dots \\ 0 & 0 & \dots & 0 & 0 & \beta_{N_{\text{thick}}+1} & \alpha_{N_{\text{thick}}+2} & \dots \\ \vdots & & & & & & & \ddots \end{pmatrix},$$

and restarts as above, repeating under convergence.

This thick-restart algorithm requires more matvec multiplications than standard Lanczos, because information is thrown away at each restart, but the storage and reorthogonalization of Lanczos vectors can be greatly reduced. There is no recommended value of N_{thick} or k , but one should take k as large as practical, and “typical” values of $N_{\text{thick}} \approx 3N_{\text{keep}}$ or so.

Although the usual application to thick restart is to find low-lying states, it is conceivable to choose a slice of excited energy and to converge excited states. This will be investigated.

7.3 Can I restart standard Lanczos?

The standard Lanczos algorithm is an iterative algorithm. In principle, if you found the desired eigenpairs had not converged under the chosen number of iterations, you could pick up and restart. To do this you would need the Lanczos vectors created so far and the Lanczos coefficients.

Although in prior version **BIGSTICK** wrote the Lanczos vectors to disk, in its current version stores all Lanczos vectors in RAM. In MPI parallelization the Lanczos vectors are stored across multiple processes. Therefore right now the restart option has been turned off. It is possible in future versions we may restore it, although it is not a high priority.

Chapter 8

Parallel computing

BIGSTICK can run many non-trivial problems on modest desktop or even laptop computers. Because problems grow exponentially, however, single-processor calculations quickly reach limits. To overcome these limits we invoke parallel processing.

Although many parts of the set up portion of the code have been parallelized, by far the most time-consuming part of the code is the matrix-vector multiplication, followed by reorthogonalization, and it is these two portions it is most important to parallelize.

For very large calculations, one needs to distribute both matvec operations (work load balance) and data (memory load balance). Operations are parallelized using both MPI (distributed memory) and OpenMP (shared memory) while data can only be distributed with MPI.

When BIGSTICK starts, it tells you how many MPI processes and how many OpenMP threads per process it is using:

```
Number of MPI processors =      512
NUM_THREADS =              8
```

This information is also written to the `.log` file. BIGSTICK does not have any special requirements for setting up parallel runs, although to run in parallel one must use an executable compiled with parallel options, i.e. `bigstick-mpi.x` compiled with `make mpi`, `bigstick-omp.x` compiled with `make openmp`, or the hybrid `bigstick-omp-mpi.x` compiled with `make openmp-mpi`. Any user who wishes to use the parallel capability should already have some idea about submitting parallel jobs. For example, to set up the number of OpenMP threads on a desktop machine you typically

```
PROMPT>export OMP_NUM_THREADS=8
```

and to submit an MPI job you may do

```
PROMPT>mpirun -n 512 bigstick-mpi.x
```

Of course, the details will depend upon the local environment. Unfortunately, in our experience supercomputers do not have a uniform job submission protocol.

8.1 MPI

To carry out a Lanczos iteration, which includes a matvec followed by reorthogonalization, one needs the following data:

- an initial vector;
- an final vector;
- jump information used for on-the-fly construction of the many-body matrix elements; and
- the uncoupled two- (or, optionally, three-) body matrix used in construction of the many-body matrix elements;
- previously computed Lanczos vectors (used for calculation of the Lanczos coefficients, for reorthogonalization and, ultimately, construction of the final eigenvectors which represent wavefunctions).

In large calculations some or all of these may need to be distributed via MPI.

To compute the distribution efficiently, **BIGSTICK** goes through the setup in two stages. First, it calculates the number of operations in each matvec, that is, in

$$v_i^{\text{final}} = \sum_j H_{ij} v_j^{\text{initial}}, \quad (8.1)$$

each update

$$v_i^{\text{final}} \leftarrow v_i^{\text{final}} + H_{ij} v_j^{\text{initial}}. \quad (8.2)$$

Because of factorization, **BIGSTICK** does not have to actually generate every operation. **BIGSTICK** then generates the distribution, and each MPI process creates locally the data it needs.

BIGSTICK attempts to distribute the operations across MPI processes as evenly as possible. The operations are constructed from jumps, but the ratio of operations to jumps is not fixed. We find it helpful to think of matvec operations as represented by the area of a rectangle, and the sides of the rectangle representing the jumps. If the rectangle is nearly square, the reconstruction is efficient, but if one has a long, thin rectangle in either dimension, one requires considerably more storage of jumps relative to the number of operations. Occasionally, an equitable distribution of operations will, on some small subset of MPI processes, so many jumps they cannot be stored. In that case, **BIGSTICK** will distribute those jumps over multiple MPI processes; this of course leads to a load imbalance, but is necessary so as not to exhaust memory.

8.1.1 Fragments

If the basis dimension is so large both initial and final vectors cannot be contained in core memory, they must be broken into *fragments*. When running in MPI, or in modeling mode, **BIGSTICK** will ask for this automatically:

```
Enter desired limit on fragment size for breaking Lanczos vectors
(Largest un-splittable block =                16602236 )
( Default =                200000000 )
( Enter 0 to use default )
```


Somewhat counterintuitively, **BIGSTICK** achieves best efficiency when the fragments are as large as possible. The reason for this is that the factorization principle behind **BIGSTICK** works most efficiently when combining large conjugate data.

Fragments are generally combinations of contiguous sectors (a portion of the vector which is labeled by the proton quantum numbers), although, because the lengths of sectors can vary significantly, in some cases a fragment can be comprised of a single sector. In the most extreme cases **BIGSTICK** will seek to divide a sector into two new ‘sectors,’ although there are limitations to how finely this can be done. Otherwise **BIGSTICK** attempts to make the fragments as similar in size as practical.

If you run **BIGSTICK** in MPI mode, it will ask for the fragment size. The fragment size is approximately the length of the initial and final vectors stored on a given MPI process (because of the way the code chunks data, **BIGSTICK** actually allows for a small overrun). Choosing a value of 0 will select the default value, currently around 200 million, which is actually on the small size. Because Lanczos vectors are stored in single precision, this requires roughly 1.6 Gb of RAM for the initial and final vector fragments. On many machines you can choose this to be larger.

Matvec operations are now defined from an initial fragment (of a Lanczos vector) to a final fragment (of a Lanczos vector). This work will generally be spread across multiple MPI processes; hence one needs `nproc` (the number of MPI processes) \geq `nfragments`² (the number of fragments). In fact, **BIGSTICK** will complain if `nproc` $< 2 \times$ `nfragments`².

8.1.2 Opbundles and optypes

The operations are organized by a derived **type** (Fortran’s designation for a bundle of data, very much like a **struct** in C) called *opbundles*, or bundles of operations. Opbundles are the ‘natural’ way to divide up work in **BIGSTICK**. Opbundles orchestrate the application of matvec operations, and **BIGSTICK** provide information about opbundles. Most users will not need this information.

Each opbundle has an associated ‘optype,’ which classifies the physical origin of the matrix elements being reconstructed. For example, the ‘PP’ optype is for interactions between two protons, with neutrons as spectators. There are also NN and PN optypes, and for three-body forces, PPP, NNN, PNN, and PPN. Finally there has been an optype SPE for single-particle energies and related single-particle potentials. However this has been absorbed into PP, PN, and NN optypes. We do this by multiplying any one-body term by $(\hat{N} - 1)/(A - 1)$, with \hat{N} the number operator and A the (valence) mass number. In the same way, if one runs with three-body forces, any two-body forces are subsumed into three-body by multiplying the two-body operators by $(\hat{N} - 2)(A - 2)$.

Optypes signal different operations and invoke different methods of reconstructing the matrix elements. PP optypes use proton ‘two-body jumps’ and loop over spectator neutron Slater determinants, NN optypes use neutron two-body jumps and loop over spectator proton Slater determinants, and PN op-

types use both proton one-body jumps and neutron one-body jumps. Not only do these invoke different subroutines, the time per operation is different for different optype, because the loops are different, and may be different on different machines. This information in turn is used to calculate the distribution of work. Information on the timing of these operations is found in the file `timinginfo.bigstick`. In many cases, by carrying out a short run to establish the time per operation, written to `timinginfo.bigstick`, and then running the desired run using this information, can lead to significantly greater efficiency.

Unfortunately, the time per operation is not as fixed for an optype as we originally hope, and detailed investigations show a great deal of fluctuations. We are still investigating this issue and attempt to arrive at better weighting and distribution algorithms.

8.1.3 Modeling

One menu option BIGSTICK offers is modeling, or choice ‘m’ on the main menu. This will run mostly like a normal run, with the following differences:

- No interaction file information will be requested (although if three-body forces are enabled, it will ask if you want to model the use of three-body forces);
- Prompt for mandatory information on fragments;
- Prompt for mandatory information on the number of MPI processes; information on the number of OpenMP threads is not needed;
- Prompt for the number of Lanczos vectors.

You can model a run using a different number of MPI processes than the modelled number.

The modeling option will calculate the distribution of work and data. This is useful because you can find out if the number of MPI processes requested is insufficient, or if BIGSTICK can find a distribution solution at all. (In some rare cases the algorithm currently fails.)

8.2 OpenMP

BIGSTICK uses OpenMP where it can, in particular in matvec. Unfortunately due to the nature of the problem, there are limitations to the speedup from OMP. Because the matrix elements are very sparse, one tends to lose locality. Modern computers have at least three levels of storage: disk storage, RAM storage, and cache storage. These three kinds of memory are increasingly close to the CPU and thus are increasingly faster; they are also increasing smaller in size. When data is fetched from disk or even from RAM, the CPU also fetches nearby data and leaves it in the cache. If the program needs that cached data next, it is handily nearby and thus faster to be accessed. Because of the highly

nonlocal nature of the data, however, **BIGSTICK** has trouble reaching maximum efficiency. While we continue to work on this issue, by the very nature of the sparse matrix this is difficult.

Chapter 9

Timing

In order to improve efficiency, BIGSTICK contains a number of built-in variables and routines for tracking and reporting timing. When running in serial, BIGSTICK uses the FORTRAN routines `date_and_time` or `cpu_time`. Unfortunately these do not provide very accurate timing, on the order of 0.01 second, so some information is not accurate. When running in MPI, BIGSTICK uses `BMPI_Wtime`, which is much more accurate.

BIGSTICK will give an estimate of the time to run,

```
Approximate time per iterations estimated : 2112 sec, or 35.2 min
```

but keep in mind this is a rough estimate. This uses information in `timinginfo.bigstick` which contains timing from previous runs. If you previously ran a similar problem, this estimate is likely reliable, but if the problem changes, or if you are using the default assumption, when `timinginfo.bigstick` does not exist, then the results may vary.

9.1 Mode times

The main timing in BIGSTICK is to measure the amount of time the code spends in various modes of operation, i.e., in generating the basis, computing jumps, matvec (matrix-vector multiplication), reorthogonalization, and so on. At the end of a run, BIGSTICK prints out the cumulative time. These times are written to the terminal as well as the `.res` results file. The output looks something like this:

```
Total time to run : 58.7889999998733
Time to compute basis : 3.999999724328518E-003
Time to count up jumps : 1.099999994039536E-002
Time to create descendents : 3.000000026077032E-003
Time to set up for matrix elements : 1.000000163912773E-003
Time to decouple matrix elements : 1.999999862164259E-003
Time to compute jumps : 1.899999985471368E-002
```

```

Time to sort 1b-jumps : 1.099999994039536E-002
Time to set up pivot : 0.000000000000000E+000
Time to compute lanczos : 42.02500000003725
Time total in H mat-vec multiply : 30.9959999998100
Time to apply sp energies : 4.599999962374568E-002
Time in pn : 13.1739999908023
Time in pn(back) : 8.17700000526384
Time in 2-body (pp) : 2.46199999982491
Time in 2-body (pp)(back) : 2.45600000442937
Time in 2-body (nn) : 4.66199999954551
Time in reorthogonalization : 10.8760000029579
Time to compute J^2, T^2 : 9.499999973922968E-002
Time in applyobs : 0.950999999884516
Time spent diagonalizing. : 7.299999939277768E-002

```

9.2 Timing for parallel runs

In addition to timing various modes during a run, **BIGSTICK** provides timing data useful for load balancing MPI parallel runs. As discussed elsewhere, **BIGSTICK** attempts to distribute work across MPI processes by counting up the number of operations and distributing the work. Operations are managed by opbundles, and each opbundle is associated with a particular Hamiltonian mode: proton-proton (PP), neutron-neutron (NN), proton-neutron (PN), and so on. Therefore **BIGSTICK** tracks the time spent on each MPI process, on each Hamiltonian mode on each MPI process, and finally on each opbundle.

Appendix A

Reduced matrix elements

The Wigner-Eckart theorem states that a matrix element which depends upon J_z is proportional to a Clebsch-Gordan coefficient, that is,

$$\begin{aligned}\langle J_f M_f | \hat{O}_{KM} | J_i M_i \rangle &= [J_f]^{-1} (J_i M_i, KM | J_f M_f) (J_f || \hat{O}_K || J_i) \\ &= (-1)^{J_f - M_f} \begin{pmatrix} J_f & K & J_i \\ -M_f & M_K & M_i \end{pmatrix} (J_f || \hat{O}_K || J_i)\end{aligned}\quad (\text{A.1})$$

where $(J_f || \hat{O}_K || J_i)$ is the *reduced matrix element*, which encapsulates the fundamental matrix element independent of orientation, and which in 5.1.3 is related to a sum over *all* orientations.

Eq. (A.1) can also be thought of as the definition of the reduced matrix element (and the Wigner-Eckart theorem a statement that this definition is consistent using any set of M s). Note that it is possible to have a variant definition with different pre-factors, that is, the phase and factors like $\sqrt{2J_f + 1}$ are conventions. Only the Clebsch-Gordan coefficients are dictated by the theorem. The choices of (A.1), taken from Edmonds [1996] are the most widely used ones.

The Wigner-Eckart theorem applies not just to angular momentum but any SU(2) algebra; hence one can reduce in isospin as well, and a *doubly*-reduced matrix element follows naturally:

$$\begin{aligned}\langle J_f M_f; T_f M_{Tf} | \hat{O}_{KM; TM_T} | J_i M_i; T_i M_{Ti} \rangle &= \\ \frac{(J_i M_i, KM | J_f M_f)}{[J_f]} \frac{(T_i M_{Ti}, TM_T | T_f M_{Tf})}{[T_f]} (J_f, T_f || \hat{O}_{K,T} || J_i, T_i).\end{aligned}\quad (\text{A.2})$$

Appendix B

The Hamiltonian and other operators in second quantization

Here we carefully define our operators in second quantization, that is, using fermion creation and annihilation operators and coupled up to good angular momentum. To denote generic operators $\hat{\alpha}, \hat{\beta}$ coupled up to good total angular momentum J and total z -component M , we use the notation

$$(\hat{\alpha} \times \hat{\beta})_{JM} = \sum_{m_\alpha, m_\beta} (j_\alpha m_\alpha, j_\beta m_\beta | JM) \hat{\alpha}_{j_\alpha m_\alpha} \hat{\beta}_{j_\beta m_\beta}, \quad (\text{B.1})$$

where $(j_\alpha m_\alpha, j_\beta m_\beta | JM)$ is a Clebsch-Gordan coefficient (here and throughout we use the conventions of Edmonds [1996]).

Hence we can define the general fermion pair creation operator

$$\hat{A}_{JM}^\dagger(ab) = (\hat{a}^\dagger \times \hat{b}^\dagger)_{JM} \quad (\text{B.2})$$

with two particles in orbits a and b . We also introduce the time-reverse of $\hat{A}_{JM}^\dagger(ab)$, the pair annihilation operator,

$$\tilde{A}_{JM}(cd) = -(\tilde{c} \times \tilde{d})_{JM} \quad (\text{B.3})$$

Here we use the standard convention $\tilde{c}_{m_c} = (-1)^{j_c+m_c} \hat{c}_{-m_c}$, where m_c is the z -component of angular momentum. An alternate notation is

$$\hat{A}_{JM}(cd) = \left(\hat{A}_{JM}^\dagger(cd) \right)^\dagger = (-1)^{J+M} \tilde{A}_{J,-M}(cd) \quad (\text{B.4})$$

With this we can write down a standard form for any one- plus two-body Hamiltonian or Hamiltonian-like operator, which are angular momentum scalars.

To simplify we use

$$\begin{aligned} \hat{H} &= \sum_{ab} e_{ab} \hat{n}_{ab} \\ &+ \frac{1}{4} \sum_{abcd} \zeta_{ab} \zeta_{cd} \sum_J V_J(ab, cd) \sum_M \hat{A}_{JM}^\dagger(ab) \hat{A}_{JM}(cd), \end{aligned} \quad (\text{B.5})$$

where $\hat{n}_{ab} = \sum_m \hat{a}_m^\dagger \hat{b}_m$ and $\zeta_{ab} = \sqrt{1 + \delta_{ab}}$. Here $V_J(ab, cd) = \langle ab; J | \hat{V} | cd; J \rangle$ is the matrix element of the purely two-body part of \hat{H} between normalized two-body states with good angular momentum J ; because it is a scalar it is independent of the z -component M . To make our results as broadly interpretable as possible, we also write this as

$$\begin{aligned} &\sum_{ab} e_{ab} [j_a] \left(\hat{a}^\dagger \times \tilde{b} \right)_{0,0} \\ &+ \frac{1}{4} \sum_{abcd} \zeta_{ab} \zeta_{cd} \sum_J V_J(ab, cd) [J] \left(\hat{A}_J^\dagger(ab) \times \tilde{A}_J(cd) \right)_{0,0} \end{aligned} \quad (\text{B.6})$$

where we use the notation $[x] = \sqrt{2x+1}$, which some authors write as \hat{x} ; we use the former to avoid getting confused with operators which always are denoted by either \hat{a} or \tilde{a} .

Finally we also can introduce one-body transition operators with good angular momentum rank K and z -component of angular momentum M ,

$$\hat{F}_{K,M} = \sum_{ab} F_{ab} \frac{1}{[K]} \left(\hat{a}^\dagger \times \tilde{b} \right)_{K,M} \quad (\text{B.7})$$

Here $F_{ab} = \langle a || \hat{F}_K || b \rangle$ is the reduced one-body matrix element.

B.1 Symmetries of matrix elements

Two-body matrix elements satisfy the following symmetries:

$$\begin{aligned} V_J(ab, cd) &= -(-1)^{j_a+j_b+J} V_J(ba, cd) \\ &= -(-1)^{j_c+j_d+J} V_J(ab, dc) = (-1)^{j_a+j_b+j_c+j_d} V_J(ba, dc). \end{aligned} \quad (\text{B.8})$$

Including isospin,

$$\begin{aligned} V_{JT}(ab, cd) &= -(-1)^{j_a+j_b+J+1+T} V_{JT}(ba, cd) \\ &= -(-1)^{j_c+j_d+J+1+T} V_{JT}(ab, dc) = (-1)^{j_a+j_b+j_c+j_d} V_{JT}(ba, dc). \end{aligned} \quad (\text{B.9})$$

Because we assume real-valued matrix elements, $V_{JT}(ab, cd) = V_{JT}(cd, ab)$. Although internally **BIGSTICK** has a specified order for storing matrix elements, the code can read in matrix elements in any order and with the indices a, b, c, d in any order.

Non-scalar spherical tensors should satisfy [Edmonds, 1996]:

$$\left(\hat{F}_{KM}\right)^{\dagger} = (-1)^M \hat{F}_{K,-M}. \quad (\text{B.10})$$

For non-charge-changing transitions, Eq. (B.10) implies $F_{ab} = (-1)^{j_a - j_b} F_{ba}^*$.

Appendix C

Highlighted references

There are a number of books and review articles on the configuration-interaction shell model. We focus on those in nuclear physics. One of the best, but nowadays difficult to get, is Brussard and Glaudemans [1977]. Some other useful references, in historical order, are [De-Shalit and Talmi, 2013], Towner [1977], Lawson and Lawson [1980] (thorough, but be aware his phase conventions differ from most others), Talmi [1993], Heyde [1994], Suhonen [2007], and others. A particular useful review article touching on many of the ideas here Caurier et al. [2005]; the review article Brown and Wildenthal [1988] is older but has useful information on applications of the shell model. The no-core shell model and other *ab initio* methods are a rapidly evolving field, but good overviews of the topic are Navrátil et al. [2000] and Barrett et al. [2013].

For angular momentum coupling a widely used reference is the slim volume by Edmonds [1996]. If you can't find what you need in Edmonds, you can almost certainly find it in Varshalovich et al. [1988]. Sadly, neither are good pedagogical introductions to the topic of angular momentum algebra.

Bibliography

- Bruce R Barrett, Petr Navrátil, and James P Vary. Ab initio no core shell model. *Progress in Particle and Nuclear Physics*, 69:131–181, 2013.
- B. A. Brown and W. D. M. Rae. The Shell-Model Code NuShellX@MSU. *Nuclear Data Sheets*, 120:115–118, 2014.
- B. A. Brown and B. H. Wildenthal. Status of the nuclear shell model. *Annual Review of Nuclear and Particle Science*, 38:29–66, 1988.
- B.A. Brown, A. Etchegoyen, and W.D.M. Rae. Computer code OXBASH: the Oxford University-Buenos Aires-MSU shell model code. *Michigan State University Cyclotron Laboratory Report No. 524*, 1985.
- P.J. Brussard and P.W.M. Glaudemans. *Shell-model applications in nuclear spectroscopy*. North-Holland Publishing Company, Amsterdam, 1977.
- E. Caurier and F Nowacki. Present status of shell model techniques. *Scopus Preview*, 30:705–714, 1999.
- E. Caurier, G. Martínez-Pinedo, F. Nowacki, A. Poves, J. Retamosa, and A. P. Zuker. Full $0\hbar\omega$ shell model calculation of the binding energies of the $1f_{7/2}$ nuclei. *Phys. Rev. C*, 59:2033–2039, Apr 1999. doi: 10.1103/PhysRevC.59.2033. URL <http://link.aps.org/doi/10.1103/PhysRevC.59.2033>.
- E. Caurier, G. Martinez-Pinedo, F. Nowacki, A. Poves, and A. P. Zuker. The shell model as a unified view of nuclear structure. *Reviews of Modern Physics*, 77:427–488, 2005.
- Amos De-Shalit and Igal Talmi. *Nuclear shell theory*, volume 14. Academic Press, 2013.
- T. Dytrych, K. D. Sviratcheva, C. Bahri, J. P. Draayer, and J. P. Vary. Dominant role of symplectic symmetry in *ab initio* no-core shell model results for light nuclei. *Phys. Rev. C*, 76:014315, Jul 2007a. doi: 10.1103/PhysRevC.76.014315. URL <http://link.aps.org/doi/10.1103/PhysRevC.76.014315>.

T Dytrych, K D Sviratcheva, C Bahri, J P Draayer, and J P Vary. Highly deformed modes in the ab initio symplectic no-core shell model. *Journal of Physics G: Nuclear and Particle Physics*, 35(9):095101, 2008a. URL <http://stacks.iop.org/0954-3899/35/i=9/a=095101>.

T Dytrych, K D Sviratcheva, J P Draayer, C Bahri, and J P Vary. Ab initio symplectic no-core shell model. *Journal of Physics G: Nuclear and Particle Physics*, 35(12):123101, 2008b. URL <http://stacks.iop.org/0954-3899/35/i=12/a=123101>.

Tomáš Dytrych, Kristina D. Sviratcheva, Chairul Bahri, Jerry P. Draayer, and James P. Vary. Evidence for symplectic symmetry in *Ab Initio* no-core shell model results for light nuclei. *Phys. Rev. Lett.*, 98:162503, Apr 2007b. doi: 10.1103/PhysRevLett.98.162503. URL <http://link.aps.org/doi/10.1103/PhysRevLett.98.162503>.

Alan Robert Edmonds. *Angular momentum in quantum mechanics*. Princeton University Press, 1996.

Jutta Escher and Jerry P Draayer. Electron scattering and multi- ω correlations. *Physical review letters*, 82(26):5221, 1999.

Kris LG Heyde. The nuclear shell model. In *The Nuclear Shell Model*, pages 58–154. Springer, 1994.

Calvin W. Johnson, W. Erich Ormand, and Plamen G. Krastev. Factorization in large-scale many-body calculations. *Computer Physics Communications*, 184:2761–2774, 2013.

PJ Knowles and NC Handy. A new determinant-based full configuration interaction method. *Chemical physics letters*, 111(4-5):315–321, 1984.

K D Launey, A C Dreyfuss, J P Draayer, T Dytrych, and R Baker. Emergence of cluster structures and collectivity within a no-core shell-model framework. *Journal of Physics: Conference Series*, 569(1):012061, 2014. URL <http://stacks.iop.org/1742-6596/569/i=1/a=012061>.

RD Lawson and RD Lawson. *Theory of the nuclear shell model*. Clarendon Press Oxford, 1980.

P Navrátil, JP Vary, and BR Barrett. Large-basis ab initio no-core shell model and its application to ^{12}C . *Physical Review C*, 62(5):054311, 2000.

T Neff. Hites 2012: 'horizons of innovative theories, experiments, and supercomputing in nuclear physics'. *Journal of Physics: Conference Series*, 403(1):012028, 2012. URL <http://stacks.iop.org/1742-6596/403/i=1/a=012028>.

Nuclear Shell Model. <https://www.jyu.fi/fysiikka/en/research/accelerator/nuctheory/Research/ShellModel/>

- Beresford N Parlett. *The symmetric eigenvalue problem*, volume 7. SIAM, 1980.
- William H Press, BP Flannery, SA Teukolsky, and WT Vetterling. *Numerical recipes in fortran*. Cambridge university press, 1992.
- Noritaka Shimizu. Nuclear shell-model code for massive parallel computation,” kshell”. *arXiv preprint arXiv:1310.5431*, 2013.
- P. Sternberg, E. Ng, C. Yang, P. Maris, J.P. Vary, M. Sosonkina, and H. Viet Le. Accelerating configuration interaction calculations for nuclear structure. *The Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.
- Jouni Suhonen. *From Nucleons to Nucleus: Concepts of Microscopic Nuclear Theory*. Springer Science & Business Media, 2007.
- Igal Talmi. *Simple models of complex nuclei*. CRC Press, 1993.
- J. Toivanen. Efficient matrix-vector products for large-scale nuclear shell-model calculations. <http://inspirehep.net/record/728378/>, 2006.
- Ian Stuart Towner. A shell model description of light nuclei. 1977.
- Dmitrii Aleksandrovich Varshalovich, Anatolij Nikolaevič Moskalev, and Valerii Kel’manovich Khersonskii. *Quantum theory of angular momentum*. World scientific, 1988.
- R. R. Whitehead, A. Watt, B. J. Cole, and I. Morrison. Computational methods for shell model calculations. *Advances in Nuclear Physics*, 9:123–176, 1977.