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

Calvin W. Johnson[2], W. Erich Ormand[3], Kenneth S. McElvain[4], Ryan Zbikowski [5], and Hongzhang Shan[6]

October 3, 2025

[1]UCRL number: LLNL-SM-739926

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

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

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

[5]Computational Science Research Center, San Diego State University, 5500 Campanile Drive, San Diego CA 92182

[6]Computational 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, static and transition 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. This manual also provides some, though not all, details into the inner workings.

This code is distributed under the MIT Open Source License. The source code and sample inputs are found at `github.com/cwjsdsu/BigstickPublick`.

Contents

1 Introduction **5**
- 1.1 Expectations of users . . . 6
- 1.2 How to cite and copyright notices/licenses . . . 6
  - 1.2.1 LAPACK copyright notice . . . 7
- 1.3 Reporting bugs and other issues . . . 8
- 1.4 What's new in each version . . . 9
- 1.5 A brief history of BIGSTICK, and acknowledgements . . . 9
- 1.6 This version . . . 10

2 How we solve the many-body problem **11**
- 2.1 Matrix formulation of the Schrödinger equation . . . 12
- 2.2 Representation of the basis . . . 14
  - 2.2.1 Factorization of the basis . . . 16
- 2.3 The Lanczos algorithm and computational cost . . . 19
- 2.4 Representation of the Hamiltonian . . . 20
- 2.5 An incomplete survey of other codes . . . 21

3 Getting started with BIGSTICK **24**
- 3.1 What can BIGSTICK do? . . . 24
- 3.2 Downloading and compiling the code . . . 25
  - 3.2.1 Directory structure . . . 25
  - 3.2.2 Compilation . . . 25
- 3.3 Required input files . . . 26
- 3.4 Running the code . . . 27
- 3.5 Some sample runs . . . 31
- 3.6 Typical run times . . . 31

4 Using BIGSTICK, in detail **33**
- 4.1 Overview of input files . . . 33
- 4.2 Defining the model space . . . 34
  - 4.2.1 Particle-hole conjugation . . . 38
  - 4.2.2 Truncation of the many-body space . . . 39
  - 4.2.3 Advanced truncation options . . . 42
  - 4.2.4 How to handle 'different' proton-neutron spaces . . . 42

4.3 Interaction files . . . 44
4.3.1 Scaling and autoscaling . . . 46
4.3.2 Proton-neutron and other isospin-breaking formats . . . 47
4.3.3 General one-body interactions . . . 50
4.3.4 `MFDn` format input . . . 51
4.3.5 Using `NuHamil` . . . 53
4.3.6 Three-body forces . . . 53
4.4 Primary runtime options . . . 54
4.4.1 Autoinput . . . 54
4.4.2 Standard or normal runs . . . 54
4.4.3 One-body density matrices and occupations . . . 54
4.4.4 Single-particle occupations . . . 56
4.5 Other primary options . . . 56
4.5.1 Modeling . . . 57
4.5.2 Traces . . . 57
4.5.3 Configurations and configuration occupations . . . 57
4.6 Diagonalization options . . . 59
4.6.1 Convergence . . . 61
4.6.2 Block Lanczos . . . 62
4.7 Secondary runtime options . . . 63
4.7.1 Expectation value . . . 64
4.7.2 Matrix elements of a scalar one+two-body operator . . . 65
4.7.3 Projection of states of good angular momentum . . . 66
4.7.4 Combining (and orthogonalizing) wave functions from several files . . . 66
4.7.5 Applying a one-body transition operator . . . 68
4.7.6 Applying a two-body body scalar operator . . . 69
4.7.7 Two-body transition densities . . . 70
4.7.8 Generating strength function distributions . . . 70
4.7.9 Overlap or dot product of wave functions . . . 72
4.8 Output files . . . 73
4.8.1 Secondary files . . . 74
4.8.2 Diagnostic files . . . 74
4.9 Memory usage . . . 74

5 Applications **76**
5.1 One-body density matrices . . . 76
5.1.1 Symmetries of density matrix elements . . . 78
5.1.2 Particle occupations from densities . . . 78
5.1.3 Conversion from proton-neutron to isospin . . . 78
5.1.4 Strengths from density matrix elements . . . 79
5.1.5 Sample case: spin-flip . . . 80
5.1.6 Charge-changing transitions . . . 82
5.1.7 Sample case: ^{19}F . . . 83
5.1.8 Transitions utilities . . . 83
5.2 Two-body densities . . . 83

5.3 Strength function option . . . 87
5.3.1 Decomposition . . . 88
5.3.2 Transition strength function distributions: the basics . . . 90
5.3.3 Transitions with good angular momentum . . . 94
5.3.4 Gamow-Teller with strength function option . . . 96
5.4 Resolvent/Green's function . . . 98

6 A peek behind the curtain 99
6.1 A normal run . . . 99
6.2 Writing out the basis . . . 100
6.2.1 Alternate information on basis . . . 102
6.3 Writing out the Hamiltonian and other operators . . . 106

7 Lanczos algorithm 107
7.1 Standard Lanczos algorithm . . . 107
7.2 Thick-restart Lanczos . . . 109
7.2.1 Targeted thick-restart Lanczos: interior eigenvalues . . . 112
7.3 Block Lanczos . . . 112
7.3.1 Bootstrapped block Lanczos . . . 114
7.3.2 Block strength function . . . 115
7.3.3 Thick-restart block Lanczos . . . 116
7.4 Can I restart standard Lanczos? . . . 117

8 Parallel computing and timing 118
8.1 MPI . . . 119
8.1.1 Fragments . . . 119
8.1.2 Block Lanczos . . . 120
8.1.3 Opbundles and optypes . . . 121
8.1.4 Jump storage and 'greedy' opbundles . . . 122
8.1.5 Modeling . . . 123
8.2 OpenMP . . . 123
8.3 Timing . . . 124
8.3.1 Mode times . . . 124
8.3.2 Timing for parallel runs . . . 125

9 Recent additions 126
9.1 Density matrix output . . . 126

A Matrix elements and operators 127
A.1 Reduced matrix elements . . . 127
A.2 The Hamiltonian and other operators in second quantization . . . 128
A.3 Symmetries of matrix elements . . . 129

B A summary of options 130
B.1 Main menu . . . 130
B.2 Diagonalization menu . . . 133

C Troubleshooting **135**
C.1 Overall 135
C.2 Compilation 135
C.3 Inputs 136
C.4 Large cases and parallel computing 136
C.4.1 Proton-neutron imbalance 137
C.4.2 Important clues in a failure 138
C.5 If you want to contact us with a problem 140

D Glossary **141**

E Highlighted references **143**

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, although some details are outlined in Sec. 8.1 on MPI paralleization. Some of the terminology is explained in the glossary, Appendix D The

code itself is heavily commented. 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 (Sec. 4.3.4) and NuHamil (Sec. 4.3.5).

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.

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 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 suggest additional references in Appendix E.

1.2 How to cite and copyright notices/licenses

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 number LLNL-SM-739926, arXiv:1801.08432 (this report)

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

1.2.1 LAPACK copyright notice

We use LAPACK subroutines in our code. The following are the LAPACK copyright notices.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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 found here, you still have a problem, you may send your issue to Calvin Johnson, `cjohnson@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. We have made no attempt to adapt to a Windows environment. 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 What's new in each version

BIGSTICK is highly versioned, with a three-number-code for each version, i.e., 7.11.4 (as of this writing). This allows us to track bugs that may arise, especially those introduced accidentally. Many of the output files, such as the `.res` and `.log` files, include the version number, and if you want to report a problem, you should mention that too.

In addition, the BIGSTICK distribution includes a text file, `WHATSNEW.txt`, which lists the most recent developments in the code, as well as reflecting the historical development. You should consult it especially if you received an update.

1.5 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-interactionI 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.

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, defined by quantum numbers such as J_z and π (parity), 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 Lawerence Berkeley wrote an improved algorithm for using OpenMP in matvec operations,

and Ryan Zbikowski, a Ph.D student at SDSU's Computational Science Research Center, prototyped and implemented the block Lanczos algorithm.

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. Jordan Fox helped find some bugs, and Stephanie Lauber helped find typos and confusing statements. Mark Caprio also contributed thoughtful and clarifying feedback on the manual. Dmitriy Rodkin also identified a number of bugs, for which I am grateful.

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.

1.6 This version

`BIGSTICK` versions are given a three-number code. This manual is (mostly) up-to-date for Version 7.11.4.

Chapter 2

How we solve the many-body problem

In this chapter we discuss the principles of configuration-interaction (CI) many-body calculations [Shavitt, 1998, Brussard and Glaudemans, 1977, Brown and Wildenthal, 1988, Caurier et al., 2005, Cook, 1998, Jensen, 2017, Weiss, 1961, Löwdin, 1955, Sherrill and Schaefer, 1999], 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` ([Caurier and Nowacki, 1999]), 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.

- 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,

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

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

$$\sum_{\beta} H_{\alpha,\beta} c_{\beta} = E c_{\alpha}. \tag{2.4}$$

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

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

(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. `BIGSTICK` can use single-particle states with arbitrary radial components, as long as they 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. In the same way, l really only gives the parity of each single-particle state.

Once a single-particle basis is defined, second quantization allows us to define many-body states. 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

$$\hat{a}_{i_1}^\dagger \hat{a}_{i_2}^\dagger \hat{a}_{i_3}^\dagger \ldots |0\rangle \tag{2.6}$$

is the occupation representation of the Slater determinant of the single particle states $\phi_{i_1}, \phi_{i_2}, \ldots$. 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:

$$\hat{T} = \sum_{ij} T_{ij} \hat{a}_i^\dagger \hat{a}_j, \tag{2.7}$$

where $T_{ij} = \langle i|\hat{T}|j\rangle = \int \phi_i^* \hat{T} \phi_j$ 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 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 single-particle states to have good angular momentum).

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 A.2.

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.

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. 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 $M = m_1 + m_2 + \ldots$.

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.

Because of this, some groups use group-theoretical bases, also called *symmetry-adapted bases*, such as a *SU(3)-scheme basis* ([Draayer et al., 2012]), 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.

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.

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

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 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 or $N_{\max} = 10$ 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` ([Toivanen, 2006]), `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. (Advanced versions of Lanczos, namely thick-restart Lanczos (7.2) and block Lanczos (4.6.2) are also implemented.)

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, thousands 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.

The idea is similar to the factorization of the basis. 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 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

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

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
^{4}He	$N_{\max} = 22$	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 reconstructs the value of the matrix element. 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 OXBASH ([Brown et al., 1985]) and its successors NuShell and NuShellX ([Brown and Rae, 2014]), NATHAN ([Caurier et al., 1999]), and EICODE ([Toivanen, 2006]). There have also been attempts to use group theory to construct so-called symmetry-guided bases. The main effort is in SU(3) [Draayer et al., 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 OXBASH and NuShell. The very successful code MFDn [Sternberg et al., 2008], used primarily but not exclusively for no-core shell model calculations, stores the many-body Hamiltonian matrix elements in RAM, much faster to access 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 [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. Factorization has also been used in nuclear structure physics as a gateway to approximation schemes [Andreozzi and Porrino, 2001, Papenbrock and Dean, 2003, Papenbrock et al., 2004, Papenbrock and Dean, 2005]. The codes most widely used by people beyond their authors have been OXBASH and its successor NuShell/NuShellX, and ANTOINE. Because of their wide use, and because one of us (Ormand) heavily used OXBASH, the default formats for our input .sps and .int files are heavily modeled upon the OXBASH/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. 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 few assumptions are made.

BIGSTICK allows for two species of fermions, such as protons and neutrons. 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?

BIGSTICK 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 scalar 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/BigstickPublick/
```

3.2.1 Directory structure

With more recent versions of BIGSTICK, the distribution includes the following folders/subdirectories:

```
bin             doc             make            src
```

plus a README.dat file. The src directory contains all the source code. The doc directory contains this manual and the WHATSNEW.txt file, plus others. The makefile is in the make directory. The generated executables are put in the bin directory.

3.2.2 Compilation

Your distribution includes a makefile. To access it, go into the make directory. 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`. These executables are created in the `bin` directory. 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 7.11.4Jun 2025

 Please cite: C. W. Johnson, W. E. Ormand, and P. G. Krastev
 Comp. Phys. Comm. 184, 2761-2774 (2013);
 and C. W. Johnson, W. E. Ormand, K. S. McElvain, H.-Z. Shan
  arXiv:1801.08432 and report UCRL LLNL-SM-739926

 This code distributed under the MIT Open Source License

Running on NERSC_HOST: none, scratch_dir (*.wfn,...): .
Number of MPI processors =           1 , NUM_THREADS =            6
```

Next and most important is the main menu (see Sec. 4.4 and Appendix B for further explication)

```
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                         *
 *               OPTIONS (choose one)                                      *
 * (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 (isospin fmt)  *
 * (2) Two-body density from previous wfn (default p-n format)             *
 * (x) eXpectation value of a scalar Hamiltonian (from previous wfn)       *
```

```
* (o) Apply a one-body (transition) operator to previous wfn and write out*
* (s) Strength function (using starting pivot )                            *
* (g) Apply the resolvent 1/(E-H) to a previous wfn and write out          *
* (m) print information for Modeling parallel distribution                 *
* (l) print license and copyright information                              *
* (?) Print out all options                                                *
*                                                                          *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

Enter choice
```

The most common choice is '`(n)`' for a normal run. For a guide to the various options, see Section 4.4 and Appendix B.

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 diagonsing 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`, which tells `BIGSTICK` to open the file `sd.sps`, and read in information about the *sd* valence space. (Please be aware that in most cases one does not enter the extension, such as `.sps` or `.int`.) 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 Jz 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 )
```

Important: You *must* enter `end` to finish reading in interaction files.

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 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, lastly and only infrequently, 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 4.7.1; 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 4.7.5, 5.3.2, and 5.3.3.

3.5 Some sample runs

In the directory `examples` 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 $(\mathrm{dim})^{1.25}$ for two-body interactions, and $\approx (\mathrm{dim})^{1.5}$ for three-body forces.

Of course, running in parallel will speed up the code. To estimate the run time, divide by the number of cores used. These estimates are crude, but usually within about a factor of two.

Nuclide	space	dim	# ops	min. store	run time
^{24}Mg	*sd*	28,503	8.6M	34 Mb	5 s
^{48}Cr	*pf*	2 M	1.5 B	6 Gb	15 min
^{51}Mn	*pf*	44M	41B	160 Gb	9 hr
^{56}Fe	*pf*	500M	0.9 T	3.6 Tb	6 d
^{60}Zn	*pf*	2.3B	5T	20 Tb	35 d
^{12}C	$N_{\max}$ 6	32M	41 B	160 Gb	7 hr
^{6}Li	$N_{\max}$ 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
^{6}Li	$N_{\max}$ 16	800M	7T	27 Tb	46 d

Table 3.1: 'Typical' run times for various nuclides, running *in serial* for 150 Lanczos iterations. To get approximate speed-up in parallel modes, divide by the number of cores. 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. Mandatory 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` (preferred) or `.sp` ('legacy' from NuShellX inputs). 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, which can be generated by the NuHamil code (4.3.5).. 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 to expect the MFDn format. Go to section 4.3.4 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 4.7.5, with advanced instruction and examples in sections 5.3.2 and 5.3.3.

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 been 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. You only need to define the model space when carrying out a 'primary' option.

The single-particle space is defined one or two ways. Either 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, using the autofill or 'auto' option.

For consistency, we generally refer to *orbits* as single-particle spaces labeled by angular momentum j but not j_z, while *states* are labled 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 an exclamation point ! or a hash mark # 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 intepret '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, described 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.)

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 where protons and neutrons can have different weights. The preferred format is `pnw`, where one lists the quantum numbers as well as the proton and neutron weights in two columns:

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

For some more details on using the `pnw` format, especially in delineating different proton and neutron valence spaces, see Section 4.2.4 below.

An alternate, older (and no longer recommended) format is, `wpn`, where first proton, then neutron orbits are listed in order.

```
wpn
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. (In older versions one had to list the number of both proton and neutron orbitals, but by default these now must be the same.)

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 `.sps` 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 isospin symmetry. (Note that, however,

because BIGSTICK requires the proton and neutron spaces to be the same, one does not need this option, and independent of the form of the single-particle space file one can read in interaction matrix elements in either isospin-conserving or -breaking format.) 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 differentce 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 allow 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 weighting 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 *autofill* option for defining the single-particle states, by entering auto in place of the name of the .sps file:

```
  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., that is, for given N, in order of increasing j, 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. So, for example, if one choose the principle quantum number $N = 5$ this includes up to the $2p$-$1f$-$0h$ shells, which will looks like

```
iso
21
0.0  0.0  0.5  0
0.0  1.0  0.5  1
0.0  1.0  1.5  1
1.0  0.0  0.5  2
0.0  2.0  1.5  2
0.0  2.0  2.5  2
...
1.0  3.0  3.5  5
0.0  5.0  4.5  5
0.0  5.0  5.5  5
```

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. To understand this, , 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 the timing difference can become quite dramatic.

Note that if you want to completely fill a space (a *plenum* rather than a vacuum), for example, to have all the neutron orbits filled, you should enter in the maximum valence number; as long as the flag `iffulluseph` in module `bmodule_flags.f90` is set to `.TRUE`, the code will automatically convert it to particle-hole. This will work with if you have a truncation and set $W = 0$; this can be used if you want to force protons and neutrons to be in different spaces. I would advice against trying this with a nontrivial truncation ($W > 0$).

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 (e.g., SU(2) for the J-scheme, or the symmetry-adapted SU(3) scheme), but that is beyond the scope 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 addititive quantum number: each basis state has a total W which is the sum of the individual ws 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 ws, which would break isospin, but we haven't explored this in depth).

Given the basis parameters, the single-particle orbits and their assigned ws 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 trunction 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 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

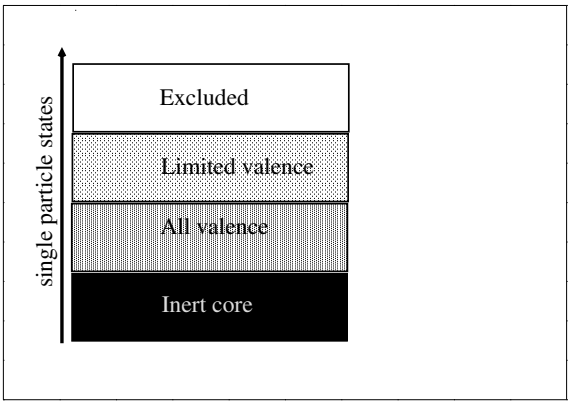

Figure 4.1: Segregation of single-particle space. 'Inert core' has all states filled. 'Excluded' disallows any occupied states. 'All valence' can have states up to the number of valence particles filled, while 'Limited valence' can only have fewer states filled (e.g. one, two, three...). See text for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

function of laboratory coordinates, $\Psi = \Psi(r_1, r_2, r_3, \ldots)$. 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:

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

(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 (see [Palumbo, 1967, Palumbo and Prosperi, 1968, Gloeckner and Lawson, 1974]):

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

This is sometimes called the $N_{\rm max}$ 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 ws. The assigned ws 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 '`pnw`' format:

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

The dimensions of the proton and neutron orbits must be the same, as the order of all the quantum numbers besides w. The values of the ws can be different for proton and neutron orbits, however, as above.

It is possible to get a more fine-grained truncation. When asked,

```
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. Please note, however, this truncation may not be robust for post-processing options such as expectation values (option '`x`') and strength functions ('`s`'), so we recommend avoiding this option.

4.2.4 How to handle 'different' proton-neutron spaces

As of the current version, `BIGSTICK` cannot directly handle independently defined proton and 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 responsiblity 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 more general, and **recommended**, approach is to use the `pnw` format: suppose you want protons active in $0f_{7/2}$, $1p_{3/2}$ and $1p_{1/2}$, and neutrons in $1p_{3/2}$, $1p_{1/2}$, and $0f_{5/2}$. Set up the .sps file

```
pnw
4
       0.0  3.0  3.5  0    0
       1.0  1.0  1.5  0   99
       1.0  1.0  0.5  0   99
       0.0  3.0  2.5 99   99
```

It is required that the proton and neutron orbits be the same, though the weight factors w is the last column can differ. A weight of 99 signals that the orbital is 'sterile' for either protons or neutrons, which means it will not be used. 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. For example, setting

```
pnw
4
       0.0  3.0  3.5  0     99
       1.0  1.0  1.5  1     99
       1.0  1.0  0.5  1     99
       0.0  3.0  2.5  99     0
```

and setting the maximum truncation to 2, you can excite up to 2 protons out of the $0f_{7/2}$ into the $1p_{3/2}$ and $1p_{1/2}$ orbits, but none into the $0f_{5/2}$, while you will have only neutrons in the $0f_{5/2}$ but none in the $0f_{7/2}$-$1p_{3/2}$ -$1p_{1/2}$ orbits.

Here you must carefully consider the nature of the proton-neutron interaction. Suppose you wanted four valence protons in the $0f_{7/2}$ -$1p_{3/2}$ - $1p_{1/2}$ space and 2 neutrons in the $0f_{5/2}$. You could also set

```
pnw
4
       0.0  3.0  3.5  0    0
```

```
1.0  1.0  1.5  1    0
1.0  1.0  0.5  1    0
0.0  3.0  2.5  99  99
```

Because the $0f_{7/2}$ -$1p_{3/2}$ - $1p_{1/2}$ space has a total of 14 states, you have have instead set valence $N = 14+2 = 16$. With `max excite` = 2, the neutrons in the $0f_{7/2}$ -$1p_{3/2}$ - $1p_{1/2}$ space will be fixed. In such cases it is often more efficient to use particle-hole conjugation (section 4.2.1). In the valence neutron $0f_{5/2}$ space one wants 2 valence neutrons or 4 neutron holes, one then sets the number of valence neutrons to -4. `BIGSTICK` will confirm this corresponds to 16 neutrons all together, although it is not clever enough to tell you that 14 of them are fixed in a closed core.

In this example, while the neutrons in $0f_{7/2}$ -$1p_{3/2}$ - $1p_{1/2}$ are fixed, they can have matrix elements with other particles, producing a change in single-particle energies. You should therefore understand carefully both your model space and your interactions.

Important: Be careful in how you read in your interaction file. Although you are treating the proton and neutron spaces separately, in many cases the supplied interaction file, at least for empirical valence spaces, will still be in `iso` format (see the next section for detail). You can test this by trying a small cases in your space, for example, just two protons and two neutrons. If you have set up correctly, you will get integer values of J. Alternately, if you get irrational values of J, the most likely culprit is that you have put in the wrong format for the interaction file.

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 `OXBASH/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 ith orbit. (Note: older version required only 10 single particle energies are on each line. This has been changed and is no longer required.) As a check, however, you should confirm that the code is reading in the correct first two-body matrix element, which is written both to screen and to the log file:

```
As a check, first two-body matrix element is   -1.37960005
```

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

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

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 interaction files.**

Important: `Ntbme` cannot be 0. If it is zero, then `BIGSTICK` will assume there are no matrix elements. Some `NuShell` input files have a zero here, but that will cause a problem. `BIGSTICK` will give a warning:

```
NO TWO-BODY MATRIX ELEMENTS FOUND
```

Note, however, that sometimes you might want to have no two-body matrix elements, for example, to add in single-particle energies only.

You can, however, set `Ntbme` larger than the actual number of two-body matrix-elements, and `BIGSTICK` will recover gracefully when it reaches the end of the file.

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

$$\left(\frac{A_0}{A}\right)^x \tag{4.3}$$

where A_0 is the reference mass number (typically A of the frozen core +2, as it is fit to the interaction of two particles above the frozen core), 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: spescale, A0,A,X
( spescale scales single particle energies,
 while TBMEs are scaled by (A0/A)^X ) for TBMEs
 (If A or X = 0, then TBMES scaled by A0 )
```

Typically the single particle energies are unscaled, but we allow for it. A typical entry, for example for the USDA/B interactions ([Brown and Richter, 2006]), would be

```
1   18   24   0.3
```

Here the single particle energies are unscaled, the core has mass number 16 and hence the reference mass A_0 is 18, the target mass in this case has mass number $A = 24$, 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 $A_{\rm core}$, 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.

Keep in mind that not all interactions will be scaled. *Ab initio* interactions are almost never scaled, and 'phenomenological' interactions depend on how they were derived and fit. See your interaction provider for more information.

If you enable autoscaling (by setting the number of matrix elements negative) and set the three parameters ($A_{\rm core}$, reference mass, and exponent) to zero, i.e., so it looks like

```
 -63     1.9798   -3.9436   -3.0612   0    0     0
```

then all parameters will be left unchanged, that is, autoscaled by one; furthermore, you will not be asked to enter in scaling factors. Autoscaling in both forms may be useful for impatient users and and users not comfortable with scaling.

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 4.3.4, 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 and neutron labels in the second and fourth columns, that is, for $V_J(ab, cd)$, a and c *must* be proton labels and b, d must be neutron labels. With twice as many defined orbits, one must also provide separate 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

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

(*Note: In older versions of this manual, the ratio in (4.4) was erroneously reversed. Eq. (4.4) is now consistent with (4.5) and (4.6)*) Here we have marked the orbits a, c as proton and b, d as neutron, but the Kronecker-δs refer only to the quantum numbers n, l, j. For example, in the *sd* shell, with the labels mentioned above,

$$V_J^{xpn}(16, 25) = \frac{1}{\sqrt{2}} V_J^{upn}(16, 25)$$

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 $d_{5/2}$

(Another wrinkle: `NuShellX`-style files *occasionally*, albeit rarely, include Hermitian conjugates of certain **proton-neutron** elements. Specifically, for matrix elements of the form $V_J^{upn}(a_\pi b_\nu, b_\pi a_\nu)$, where $a \neq b$, the matrix elements $V_J^{upn}(b_\pi a_\nu, a_\pi b_\nu)$ is also included in the file. Because `BIGSTICK` automatically fills in matrix elements using Hermiticity, and allows for any ordering of a, b, c, d, (so that if $V(ab, cd)$ is read in, then $V(cd, ab)$ is *always* automatically included); this means the code struggles to account for these extra elements. **Use such input files with caution**.)

It is up 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

$$V_J^{\rm pn}(ab,cd) = \frac{\sqrt{(1+\delta_{ab})(1+\delta_{cd})}}{2}\left[V^{\rm iso}_{J,T=0}(ab,cd) + V^{\rm iso}_{J,T=1}(ab,cd)\right], \qquad (4.5)$$

but the unnormalized convention yields the simpler

$$V_J^{\rm upn}(ab,cd) = V^{\rm iso}_{J,T=0}(ab,cd) + V^{\rm iso}_{J,T=1}(ab,cd). \qquad (4.6)$$

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.
```

That is, you must *first* enter either the code `xpn` or `upn`, and then the filename:

```
 Enter two-body interaction file name OR file format code (e.g., XPN)
 (Enter "end" to finish; "opt" for file format options; "?" for  general info )
xpn
 Enter name of two-body interaction file in explicit proton-neutron format
usdbpn
```

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 General one-body interactions

Interactions generally include one-body and two-body contributions, with three-body used for advanced applications. For historical reasons, the standard format adopted for `BIGSTICK` only reads in the the diagonal part of the one-body Hamiltonian, usually referred to as *single-particle energies.* In phenomenological spaces, such as the sd and pf spaces, this is all that is possible. In multi-shell spaces, however, one can have off-diagonal matrix elements of a one-body part of the Hamiltonian, between orbits with the same l and j but different n.

If you need to use off-diagonal one-body matrix elements, i.e. for some kind of potential which is not diagonal in the basis, there is an extension for both the `iso` (default) and the `xpn` interaction formats. In addition to the mandatory list of single-particle energies at the beginning of the file, one can add one-body matrix elements *after* reading in the two-body matrix elements (*be sure* that the number of two-body matrix elements are correctly specified at the tope of the file). Immediately after the last two-body matrix element, specify the number of one-body matrix elements.

```
N1me  ! = # of one-body potential matrix elements
```

followed by a list of the orbital indices and the one-body matrix elements.

```
   a      b    U(a,b)
```

where a, b are the orbital labels in the `xpn` ordering–hence proton and neutron labels are different–and $U(a, b)$ is the matrix element. (Note that, like the two-body matrix elements, these are *not* reduced via the Wigner-Eckart theorem.) The order of a, b is not important, and only nonzero matrix elements need to be added. You only should list a, b and not also b, a–but again, the order does not matter. Diagonal ($a = b$) are okay if they were not already read in as "single-particle energies." Hence these will look like

```
4   !  # of one-body matrix elements.
1     2    -0.0144
1     3     1.0888
2     3    -2.2220
4     5     0.8877
```

BIGSTICK will automatically check for these matrix elements. If they are not present BIGSTICK will skip over them gracefully. If you are working in the xpn format, be sure that you have included the matrix elements for both protons and neutrons; even if identical. The diagonal elements can either appear at the beginning of the file, as 'single-particle energies,' or here. If you specify the diagonal one-body matrix elements at the end of the file, then you must set the single-particle energies at the beginning of the file to zero; not listing single-particle energies, even zeroes, will confuse BIGSTICK.

Note: at this time, BIGSTICK cannot handle off-diagonal one-body Hamiltonians for a single particle *or* a single hole. The reason is, BIGSTICK generally converts the one-body part of the Hamiltonian to an effective, number-dependent two-body operator, but this does not work if there is only one particle (or, if one invokes particle-hole conjugation, as described in section 4.2.1, a single hole). For diagonal single-particle energies BIGSTICK can handle a single particle or hole with a specialized routine. In principle this could be generalized, but to date has not been.

We emphasize all of the above regards the *Hamiltonian*. One-body operators read in for option '(o)' can be completely general, i.e., non-scalar or having angular momentum rank $K > 0$: see section 4.7.5.

4.3.4 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 an MFDn-format file, you must enter the *entire* name, including any extensions. This signals to BIGSTICK to prepare to read in an an MFDn-format file. BIGSTICK will treat a file TBME.int very differently if you answer 'TBME' versus 'TBME.int' for the file name. MFDn-format files are almost always for *ab initio* or so-called *no-core shell model* calculations, and almost always assume a harmonic oscillator basis.

The input file first line is

nTBME (other stuff which are not needed)

where nTBME is the number of TBMEs in the file. For example

```
2056271     13   14    20.0000     2.0000
```

The only number BIGSTICK requires is the first one. The fourth number, 20.000, is $\hbar\Omega$ in MeV, but not all codes generate this information.

The `MFDn` format does not include explicit single-particle energies. Subsquent lines are of the form

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

or, more commonly,

```
a b c d J T Trel 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. The isospin T is not really used.

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

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

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

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

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

Here $\hat{H}_{\rm cm}$ is the center-of-mass Hamiltonian, used to push up spurious states via the Palumbo-Lawson-Glocke method ([Palumbo, 1967, Palumbo and Prosperi, 1968, Gloeckner and Lawson, 1974]):

$$\hat{H}_{\rm cm} = \frac{P^2_{\rm cm}}{2Am_N} + \frac{1}{2}Am_N\Omega^2 R^2_{\rm cm}. \tag{4.9}$$

where m_N is the nucleon mass and

$$\vec{R}_{\rm cm} = \frac{1}{A}\sum_i \vec{r}_i, \;\; \vec{P}_{\rm cm} = \frac{\hbar}{i}\sum_i \vec{\nabla}_i. \tag{4.10}$$

These have the correct commutation relation, that is, $[\vec{R}_{\rm cm}, \vec{P}_{\rm cm}] = i\hbar$, so that $\vec{P}_{\rm cm}$ is the conjugate momentum to $\vec{R}_{\rm cm}$.

It is useful to separate $\hat{H}_{\rm cm}$ into one- and two-body parts:

$$\begin{aligned}\hat{H}_{\rm cm} = &\frac{1}{2Am_N}\sum_i p_i^2 + \frac{1}{2A}m_N\Omega^2\sum_i r_i^2 + \\ &\frac{1}{2Am_N}\sum_{i\neq j}\vec{p}_i\cdot\vec{p}_j + \frac{1}{2A}m_N\Omega^2\sum_{i\neq j}\vec{r}_i\cdot\vec{r}_j\end{aligned} \tag{4.11}$$

The first two terms are the single particle energies, with values $\hbar\Omega(N+3/2)/A$, with N the principal quantum number, and the second two terms is $\hat{H}_{rel}\times\hbar\Omega/A$. `BIGSTICK` automatically accounts for all the factors, as long as you provide the correct $\hbar\Omega$ as shown below:

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 select 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 $\beta_{c.m.}$. Typical values of $\beta_{c.m.}$ are 1-10.

4.3.5 Using `NuHamil`

Takayuki Miyagi's open source code `NuHamil`, (arXiv:2302.07962, github.com/Takayuki-Miyagi/NuHamil-public) can generate a number of interactions from chiral effective field theory. One can get `NuHamil` to generate an file in MFDn format by including the line

```
ext_2bme = ".MFDn"
```

in the Python script `exe/NuHamil_2BME.py` included with the NuHamil distribution. We take no responsibility for installing or running `NuHamil`.

4.3.6 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@sdsu.edu`. We do not have the codes or capability to generate three-body forces for users.

In 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, use two-body forces 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 answering '`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 4.7. In most primary runs one solves the matrix eigenvalue problem, which invokes the diagonalization options menu, discussed in Appendix B.2.

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

$$\rho_K^{fi}(a^\dagger b) = [K]^{-1}\langle J_f||(a^\dagger b)_K||J_i\rangle \tag{4.12}$$

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

$$\langle J_f||O_K||J_i\rangle = [J_f](J_f M_f, KM|J_i M_i)^{-1}\langle J_f M_f|O_K M|J_i M_i\rangle \tag{4.13}$$

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

$$\langle \Psi_f, J_f||\hat{\mathcal{O}}_K||\Psi_i, J_i\rangle = \sum_{ab}\langle a||\hat{\mathcal{O}}_K||b\rangle\rho_K^{fi}(a^\dagger b) \tag{4.14}$$

where a, b are labels for single-particle orbits, and $\langle a||\hat{\mathcal{O}}_K||b\rangle$ are the reduced one-body matrix matrix elements for the operator $\hat{\mathcal{O}}$ with angular momentum K.

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                              *
 * (dxp) Compute one-body densities from prior run (.wfn) in p-n format.  *
```

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`.'

Three variations are the option '`dx`,' which reads in a previously computed wave function file and from it computes the one-body density matrices in isospin format; '`dxm`', which does the same but generates the density matrices in a format readable by Mathematica; and `dxp` which computes from a prior wave function file the one-body densities in proton-neutron format. The output files have the extension `.dres`. (At this time, there is not an option to write out one-body densities in a proton-neutron format readable by Mathematica.)

When invoked, these options will ask for a range of initial and final states, .e.g,

```
          5  states
  Enter start, stop for initial states
  (Enter 0,0  to read all )
0 5
  Enter start, stop for final states
  (Enter 0,0  to read all )
2 3
```

This is useful for cases where one has generated many states but only wants to extract densities for a few states.

Note: When running in MPI mode, the amount of work need to calculate one-body densities is far less than for computing the two-body Hamiltonian. BIGSTICK can get hung up when trying to distribute work for densities over a large number of MPI ranks. In that case, it is better to simply generate the wave functions using a large number of MPI ranks, and then re-run with option `dx/dxp` with few MPI ranks.

One-body densities: binary format

This is still in progess and will be updated in future versions. See Gorton [2024]

4.4.4 Single-particle occupations

```
  * (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.

When you run a density matrix option, such as '`(d)`', etc., in addition to the `.dres` file the occupations will be automatically written to a `.occres` file. The format here is more machine-readable:

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

 State #    1 E =  -87.10445 2xJ, 2xT =    0   0
   1    0.5245    0.5245
   2    3.0420    3.0420
   3    0.4335    0.4335
```

Experience suggests this is a better route in most cases than our somewhat clumsy format

4.5 Other primary options

Here we briefly discuss a number of other options from the primary menu.

4.5.1 Modeling

```
* (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. See also section 8.1.5. No interaction files are read in and no diagonalization is carried out.

The option '`(m0)`' will compute only the basis dimension, while the option '`(md)`' will compute the memory requirements for computing one-body densities in MPI.

4.5.2 Traces

There is an additional option most users are unlikely to use but which we mention nontheless.

```
* (c) Compute traces
```

`BIGSTICK` can 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.

4.5.3 Configurations and configuration occupations

Finally, the menu option '`(co)`' will, after a normal run, compute the *configuation occupation*. (The option '`(cx)`' will compute the configuration occupation from a previously computed wave function.) Here a 'configuration' is sometimes also called a 'partition,' and means a subspace defined by the occupation of orbits. That is, if we have proton orbits $0s_{1/2,\pi}$, $0p_{3/2,\pi}$, $0p_{1/2,\pi}$, etc, and similarly for neutrons, we label a subspace by the number of protons and neutrons in each orbit, e.g. $(0s_{1/2,\pi})^2(0p^3_{3.2,\pi}(0p^1_{1/2,\pi},0s_{1/2,\nu})^1(0p^2_{3.2,\nu}(0p^2_{1/2,\nu}$. The '`(co)`' option produces what fraction of the wave function is in each configuration subspace/partition.

In addition to producing the usual `.res` and `.wfn` files, a file with extension `.cfo` will contain the configuration occupations, which are in proton-neutron format. The `.cfo` begins with a description of the single-particle valence space and a list of the orbits:

```
Z =    4 N =    4    ! valence # of protons and neutrons
 Proton orbits
```

```
    0    2    3    0     ! n   l    2xj    w
    0    2    5    0
    1    0    1    0
  Neutron orbits
    0    2    3    0
    0    2    5    0
    1    0    1    0
  Parity =            1  ! note: if all particles have same parity, then parity=1
                             ! otherwise parity = 1 is +, parity = 2 is -
  No W truncation
```

It then lists the proton and then neutron 'configurations':

```
  Proton configurations
     1 :   4  0  0
     2 :   3  1  0
     3 :   3  0  1
     4 :   2  2  0
...
     8 :   1  2  1
     9 :   1  1  2
    10 :   0  4  0
...
```

where, e.g,

```
     2 :   3  1  0
```

means proton configuration #2 has 3 protons in orbit 1 (which, from the list of orbits, is the $0d_{3/2}$ orbit), 1 proton in orbit 2 ($0d_{5/2}$) and none in orbit 3 ($1s_{1/2}$) .

After the proton configurations the neutron configurations are listed in a similar manner, and then finally, for each eigenstate, the combined configurations and the fraction of the wave function in each configuration:

```
  State            1
config   p config   n config      fraction
       1        1        1        0.0000039
       2        1        2        0.0000000
       3        1        3        0.0000000
       4        1        4        0.0001510
...
     116       10        8        0.0216867
     117       10        9        0.0024511
     118       10       10        0.1866053
     119       10       11        0.0367340
     120       10       12        0.0250714
....
```

Note that for this state, 18.7% of the wave function is in proton configuration 10 and neutron configuration 10. Looking up at the list of proton configurations above, that configuration has four protons in the $0d_{5/2}$ orbit; although not shown, it is the same here for the neutron configuration.

4.6 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                              |
|                                                                          |
 | (bd) Block Lanczos with default convergence (STANDARD)                  |
 | (bf) Block Lanczos with fixed (user-chosen) iterations                  |
 | (bc) Block 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       |
 | (tb) Thick-restart block Lanczos with default convergence
 |                                                                         |
 | (sk) Skip Lanczos (only used for timing set up)                         |
 | (li) Lanczos iterations only, no further eigensolutions                 |
 |                                                                         |
 \ ------------------------------------------------------------------------/
```

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 4.6.1, is satisfied, or until 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 usualy longer.

The default convergence check is discussed in the next section, 4.6.1. If you want a fixed number of iterations without checking convergence, choose '`lf`.' If you want finer control over convergence, choose '`lc`,' discussed in 4.6.1.

Block Lanczos is discussed below in section 4.6.2, with additional information in section 7.3.

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 [Wu and Simon, 2000], which requires fewer vectors stored in memory but requires more iterations. While standard Lanczos finds the lowest $N_{\rm keep}$ eigensolutions with $N_{\rm iter}$ iterations, thick-restart has three numbers: $N_{\rm keep} < N_{\rm thick} < N_{\rm iter}$. As described more fully in section 7.2, after $N_{\rm iter}$ iterations the approximate Hamiltonian is diagonalized, and $N_{\rm thick}$ of these eigenvectors are kept for restarting. This process is repeated until convergence or until a maximum number of restarts has been exhausted.

```
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 $N_{\rm thick} >\sim 3\times N_{\rm keep}$, and we take $N_{\rm iter}$ as large as practical. Specifically, `BIGSTICK` chooses

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

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

If you want a fixed number of iterations, choose '`tf`', and you will be prompted for $N_{\rm keep}$, $N_{\rm iter}$, and then $N_{\rm thick}$:

```
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 vincinity 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. Option '`li`' carries out the Lanczos iterations but does not solve the matrices; it can be used for example, if one wants a very large number of Lanczos α and β coefficients.

4.6.1 Convergence

As `BIGSTICK` iterates, it checks for convergence. Every ten iterations it prints to screen the current $N_{\rm keep}$ 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 $N_{\rm keep}$ lowest energies but also for the next 5, and divides by the square root of $N_{\rm keep}+5$, that is,

$$\delta_{\rm conv} \equiv \frac{\sum_{i=1}^{N_{\rm keep}+5} \left| E_i^{\rm new} - E_i^{\rm old} \right|}{\sqrt{N_{\rm keep}+5}} \tag{4.15}$$

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 and block Lanczos

4.6.2 Block Lanczos

The standard Lanczos algorithm, discussed in detail in Chapter 7, applies in each iteration the Hamiltonian to a vector to create a new vector. In the block Lanczos algorithm, the Hamiltonian is applied to a block of vectors to create a block of new vectors. If used judiciously, this can improve the performance of the code.

Choosing the default option, '`bd`', will lead to the following questions

```
  Enter nkeep, dimension of blocks, max # of block iterations
  (nkeep = # of states printed out; typically ~ dim block )
```

As with standard, or vector Lanczos, `nkeep` is the number of final states desired. The dimension of the block is the number of vectors in a block, and the number of block iterations is exactly what it sounds like–the number of times one iterates to create a new block. Therefore, all things being equal, # of block iterations $\times$ dimension of block $\approx$ # of vector Lanczos iterations. For example, if one chose a block dimension of 10, that is, ten vectors in each block, and carried out 15 block iterations, that is roughly similar to $10 \times 15 = 150$ ordinary Lanczos iterations.

In general we recommend that `nkeep` $\leq$ `dim block`. The exception is for the 'block strength' option '`(sb)`' in the main menu; see Section 4.7.8.

As we discuss in detail in section 7.3, however, the actual Hamiltonian $\times$ block multiplication runs much faster than ordinary Hamiltonian $\times$ vector multiplication–up to twice as fast! The downside is that, if one just uses a random pivot, one requires significantly more iterations than the equivalent vector Lanczos run. If one uses a block of pivot vectors which are good approximations, however, the number of required iterations can be dramatically reduced. For example, one could construct states in a truncated model space. We have written a tool to project vectors from a smaller space to a a larger space, although it is not yet ready for release.

To read in one or more pre-calculated pivot vectors, use option '`(np)`' in the main menu. You will be asked to enter choices from a list of pre-calculated states. First you will be asked whether you want to read in a contiguous block (e.g., states 1 through 10 or 20 through 30), or if you want to specify all the states.

```
Need to pick a set of states for the pivot block
 Choose either (c) a contiguous list or (s) list of specified states
```

If you enter '`c`', then enter the start and stop of the list. This must be within the number of states available. If this list does not fill up the pivot block, the rest will be created as random vectors, suitably orthogonalized. This is a good option if you have large block dimensions.

If you enter '`s`', then you will be asked to specify each of the vectors, i.e., `1,2,5,8, 13, ....` if you enter '0' then a random vector will be substituted.

A description of an implementation of block Lanczos with thick-restart can be found in [Shimizu et al., 2019], which includes discussion of some performance issues.

Running block Lanczos on parallel machines using MPI has additional constraints. See section 8.1.2 for information.

4.7 Secondary runtime options

Once `BIGSTICK` has generated wave function, it can further process the wave functions in secondary options. 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` reconstucts 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 (section 4.7.1), '`o`,' apply a one-body non-scalar transition operator (section 4.7.5 and Chapter 5), and '`s/sn/su`,' the strength function option (section 4.7.8 and Chapter 5).

```
* (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. A related and more widely useful option is the strength function option '`s`' discussed below and in section .

The downside is that using an initial pivot will only accelerate the convergence of a single state. Option '`(np)`' is more useful for block Lanczos, where one can read in multiple vectors; see section 7.3.1.

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

Options '`dx`', '`dxp`,' and '`dxm`' read in a previously generated wave function and compute the one-body density matrices. The latter provides a Mathematica-friendly file format. The outputs for '`dx`' and '`dxm`' are in isospin format while for '`dxp`' it is in proton-neutron format. The output files have the extension `.dres`. More details about one-body density matrices are found in section 5.1.

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

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

4.7.1 Expectation value

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

The option '`x`' allows you to compute the expectation value of a 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, along with standard requests for scaling information. The results are written both to screen and, if an output name is given, to 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
```

The reason the norm of the vector is given is that after applying a one-body transition operator, as described in the next section (4.7.5), the wave function vector may no longer be normalized.

4.7.2 Matrix elements of a scalar one+two-body operator

```
* (h) Compute matrix elements of a scalar Hamiltonian (inputs as basis)  *
```

A generalization of computing the expectation value is to compute, for a set of wave functions, the matrix elements of an arbitrary (but angular momentum scalar) one+two-body operator, that is, something that looks like a Hamiltonian. This option works very similar to the expectation value option, except the outputs are written to a file with extension `.xme`. Zero matrix elements, including and especially those ruled out by angular momentu selection, are not written to file. The output looks like

```
 State      E        Ex         J       T      par
   1    -40.47233   0.00000     0.000   0.000    1
   2    -38.72564   1.74669     2.000   0.000    1
   3    -36.29706   4.17527     4.000   0.000    1
   4    -33.77415   6.69818    -0.000   0.000    1
   5    -32.92937   7.54296     2.000   0.000    1

          1           1  -6.62337160
          2           2  -4.44876957
          3           3  -3.29423618
          4           1  -1.27810764
          4           4  -4.65299320
          5           2  0.628762603
          5           5  -4.58469582
         -1          -1   0.00000000
```

In the above, the integers refer to the labels of the states; the time reverse is not given but has the same value. The values $-1, -1$ signal the end of the file.

(This option is useful for the `PANASh` post-processing code.)

4.7.3 Projection of states of good angular momentum

```
* (jp) Project states of good J from prior wfns and normalize                *
```

This option reads in a previous wave function with states that may be a mixture of different angular momentum J and projects out and normalizes states with good J. In particular, this option assumes you have previously used option '`(o)`' (see Sections 4.7.5, 5.3.2), and applied a one-body operator with a definite angular momentum rank K. (Future work may make this more general.)

This option starts similar to many other options, first asking for the wave function file:

```
Read in prior wfns, project good J and normalize
Enter input name of .wfn file
```

which will allow for automatic construction of the basis, and the for the output file

```
Enter output name (enter "none" if none)
```

After the basis and jumps are constructed, you may select a range of vectors to project:

```
There are           10  initial wavefunctions

Enter start, stop for initial states
(Enter 0,0  to read all )
```

You need to know the angular momentum rank (e.g., 1, 2, 3...) of the operator applied to the wave functions.

```
What is the rank (i.e, angular momentum) of operator?
```

The reason is the code will then know how many Lanczos iterations to carry out for projection. Applying an operator of rank K can, on account of the triangle rule for addition of angular momentum, i.e., starting from a state with good angular momentum J_i

$$|J_i - K| \leq J_f \leq J_i + K \tag{4.16}$$

create at most $2K + 1$ different final angular momenta J_f.

4.7.4 Combining (and orthogonalizing) wave functions from several files

```
* (ro) Read in multiple files of wfns and orthonormalize                     *
```

Section 4.7.2 describes how to generate the matrix elements of a Hamiltonian or some other Hermitian, scalar, one+two-body operator between a set of wave functions, which act as a basis of a subspace. In some cases, one may want to combine multiple files of wave functions, which may not all be orthonormal, into

a single wave function file (albeit all in the same basis). This can be done with the `'(ro)'` option.

The workflow is somewhat clunky, because of `BIGSTICK`'s standard workflows. The basic steps are:

1. Enter name of first wave function file; this will cause the creation of the common basis.
2. Enter name of **output** wave function file to be created. There will be only one final wave function file.
3. Set the total number of wave functions, `nkeep`, to be read in, usually from mutliple files.
4. Select the range of wave functions to be read, e.g., 1-5 or 15-20. Choosing `0,0` will select all in the file. After being read in, the current `.wfn` file will be closed.
5. If `nkeep` wave functions have not yet been read in, open a new `.wfn` file. **These must be in the same basis as the first wave function file, and must have good angular momentum** J.
6. Select the range of wave functions to be read, e.g., 1-5 or 15-20. Repeat until `nkeep` wave functions read in.

Here's how this looks when running:

```
  Enter choice
ro
  Read multiple files of wfns and orthnormalize
  Enter input name of .wfn file
file1
 dimbasischeck=           ...
 ...
  Enter output name (enter "none" if none)
fileout

  .... Building basis ...

  How many vectors to read in?
...
  The first file has            5  wave vectors
  Enter start, stop for initial states
  (Enter 0,0  to read all )
...
  - - -  NEXT FILE - - -
  Enter input name of .wfn file
file2
...
```

```
  Enter start, stop for initial states
  (Enter 0,0  to read all )
...
...
  All vectors read in
  Next: orthonormalize!
```

After all the wave function have been read in, BIGSTICK will orthonormalize and write to the output .wfn file.

(This option is useful for the PANASh post-processing code.)

You can also combine wave functions from several files into one file, but without orthonormalization:

```
  * (ru) Read in multiple files of wfns but DO NOT orthonormalize                *
```

4.7.5 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*
```

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           1     ! J, T of transition
    1    1  -2.68328      ! a, b   < a |||  O  ||| 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 5.1.6. For transitions which change parity, one must use a basis with both parities, option 0 in the parity-selection menu.

Because transition operators are not in general unitary, the result wave function vectors are not normalized. This information is important, as it tells us about total transition strengths, also known as the non-energy-weighted sum rule.

4.7.6 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, along with scaling, and so on.

4.7.7 Two-body transition densities

BIGSTICK can now compute two-body densities. One must carry out an ordinary run to create a `.wfn` file, e.g., options such as '`n`', '`d`', etc.. Then run BIGSTICK again, choosing '`2`' on the initial menu. You will be asked for the name of the previously computed `.wfn` file, as well as the mandatory name of the output file. The resulting file will be have a extension `.den2b`. You will also be asked

```
Enter start, stop for initial states
(This is because two-body densities are large)
(Enter 0,0  to read all )
```

with a similar choice for final states.

The list of all two-body densities can be quite long and exhaustive. Two options are '`2d`' and '`2i`' which just computes "diagonal" two-body densities, where the initial and final states are the same, and only for scalar (two-body) densities. While the two-body densities in proton-neutron formalism are still written to a `.den2b` file, one can reinterpret these as expectation values of operators, written to the `.res` file.

More details are found in Section 5.2.

4.7.8 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 5.3.

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 default 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 5.3. 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 5.3.

By default, the output wave functions are normalized. This can be made explicit by using the option '`(sn)`'. Alternately, the option '`(su)`' will instead normalize the output wave functions to the input normalization $\times$ the square root of the strength. This is useful when computing strength distributions, especially after projection, as discussed in sections 5.3.3 and 5.3.4.

Alternately, if one chooses '`ss`' then *no* wave function will be written to file and the J, T of the final wave functions will not be computed. This is recommended when working in large spaces and a large number of iterations have been carried out: the wave functions (and J, T) are generally not useful and take considerable time.

Block strength. Recently, we have added a block strength option, '`(sb)`.' Here one reads in a block of several vectors and carries out block Lanczos, thus getting strength functions for multiple starting vectors simultaneously.

As with block Lanczos (see Section 4.6.2), you will be promped:

```
 Fixed block iterations ONLY:
```

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

In standard block Lanczos, one generally wants `nkeep` $\leq$ the dimension of the block. For block strength, however, much as in standard strength option, one generally wants `nkeep` $\sim$ the number of Lanczos *vectors* generated = (dim block) $\times$ (# of block iterations).

When reading in from a previously generated file (option '`np`' in the main menu), , you can must, select the states for the pivot block. You will be prompted:

```
Enter a list of  3  states for the pivot block
(Please enter in order)
```

If you enter '0' a random vector will be generated. After this it runs just as the regular strength function, albeit with producing strength distributions for multiple input vectors. **Note**: the detailed strengths may differ from the single-state run. However by carrying out a running sum you should be able to see the *distribution* is the same.

When running large cases on parallel machines using MPI, there are additional constraints; see section 8.1.2.

Finally, by choosing '`sbs`' the wave functions will not be written to file, nor J, T computed.

4.7.9 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 intial 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.

4.8 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), 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.

4.8.1 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`.

4.8.2 Diagnostic files

BIGSTICK also generates a number of diagnostic files, primarily for development, tuning, and debugging.

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

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

4.9 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,

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

where we use reduced matrix elements as defined in Appendix A.1. We use this particular definition because the reduced matrix element of a generic one-body operator is the sum of products of the density matrix elements and the reduced matrix elements, namely,

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

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:

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

When you choose the density matrix option, BIGSTICK will write to the `.dres` file (but not to screen, and no longer to the `.res` file) 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 wavefunctions; `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,

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

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

Between the listing of the spectra (i.e., energies, excitation energies, and angular momentum and isospin) and the density matrices, an ordered list of the single-particle orbitals is given, for convenience in post-processing, e.g.,

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

`BIGSTICK` 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
```

Option '`dxp`' allows one to compute one-body densities in a proton-neutron format from a previously computed wave function.

5.1.1 Symmetries of density matrix elements

A useful symmetry relation is

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

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 the orbit occupation, you can also extract this information from the diagonal one-body density matrices. The total number of particles in orbit a is

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

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

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

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

$$n_\pi(a) = \frac{\left[\frac{1}{2}\right][j_a]}{[J_i]\,[T_i]} \frac{1}{2} \left(\rho^{ii}_{K=0,T=0}(a^\dagger a) + \frac{T_z\sqrt{3}}{\sqrt{T_i(T_i+1)}} \rho^{ii}_{K,T=1}(a^\dagger a) \right), \tag{5.7}$$

$$n_\nu(a) = \frac{\left[\frac{1}{2}\right][j_a]}{[J_i]\,[T_i]} \frac{1}{2} \left(\rho^{ii}_{K=0,T=0}(a^\dagger a) - \frac{T_z\sqrt{3}}{\sqrt{T_i(T_i+1)}} \rho^{ii}_{K,T=1}(a^\dagger a) \right). \tag{5.8}$$

5.1.3 Conversion from proton-neutron to isospin

The one-body density matrices are internally in proton-neutron formalism, but can be converted to isospin formalism *if* the initial and final states have good

isospin. (This is done in the subroutine `coupled_densities` in the file `bdensities.f90`.) We choose the convention that protons have $m_t = +1/2$, while neutrons have $m_t = -1/2$, hence $M_T = (Z-N)/2$. If ρ is a one-body density, with ρ_T isospin densities with $T = 0, 1$ and ρ_{m_t} proton/neutron densities, then

$$\rho_T = \frac{[T_f]}{(T_i\, T_z, T0|T_f\, T_z)} \frac{1}{[T]} \sum_{m_t} \rho_{m_t} \left(\frac{1}{2}\, m_t, \frac{1}{2}\, - m_t | T0 \right). \tag{5.9}$$

For the special case where $T_i = T_f$, and using analytic formulas for the Clebsch-Gordan coefficients, one gets

$$\rho_{T=0} = \frac{[T_i]}{\sqrt{2}} \left(\rho^\pi + \rho^\nu\right), \tag{5.10}$$

$$\rho_{T=1} = \frac{[T_i]}{\sqrt{2}} \left(\rho^\pi - \rho^\nu\right) \frac{\sqrt{T_i(T_i+1)}}{\sqrt{3}\, T_z}. \tag{5.11}$$

Note that here for $T_z = 0$, the $\rho_{T=1}$ density matrix must vanish. Finally, one can invert to get

$$\rho^{\pi,\nu} = \frac{1}{\sqrt{2}\,[T_i]} \left(\rho_{T=0} \pm \frac{T_z\sqrt{3}}{\sqrt{T_i(T_i+1)}} \rho_{T=1} \right). \tag{5.12}$$

This agrees with Eq. (5.7), (5.8).

5.1.4 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,

$$\frac{1}{2J_i+1} \sum_{M_i} \sum_{M_f} \left| \langle J_f M_f | \hat{O}_{KM} | J_i M_i \rangle \right|^2 . \tag{5.13}$$

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-Eckart theorem to rewrite the average/sum as:

$$\frac{1}{2J_i+1} \sum_{M_f} \sum_{M_i} \sum_{M} |(J_i M_i, KM | J_f M_f)|^2 \left| (J_f || \hat{O}_K || J_i) \right|^2 . \tag{5.14}$$

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

$$\sum_{M_i} \sum_{M} |(J_i M_i, KM | J_f M_i + M_f)|^2 = 1 \tag{5.15}$$

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

$$\frac{1}{(2J_i+1)} \sum_{M_i} \sum_{M_f} \sum_{M} \left| (J_f M_f | \hat{\mathcal{O}}_{KM} | J_i M_i) \right|^2 = \frac{1}{(2J_i+1)} \left| (J_f || \hat{\mathcal{O}}_K || J_i) \right|^2 , \tag{5.16}$$

As one often calls

$$\frac{1}{2J_i+1} \left| (J_f || \hat{\mathcal{O}}_K || J_i) \right|^2 , \tag{5.17}$$

the 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 compute transition strengths using transition density matrix elements: the doubly reduced transition matrix element for a one-body operator $\hat{\mathcal{O}}_{K,T}$ of angular momentum rank K and isospin rank T is

$$\langle \Psi_f ||| \hat{\mathcal{O}}_{K,T} ||| \Psi_i \rangle = \sum_{ab} \rho^{fi}_{K,T}(ab) \ \langle a ||| \hat{\mathcal{O}}_{K,T} ||| b \rangle \ . \tag{5.18}$$

Although the default output is 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),

$$B(\mathcal{O} : i \to f) = \frac{1}{2J_i+1} \left| (\Psi_f : J_f || \hat{\mathcal{O}}_J || \Psi_i J_i) \right|^2$$
$$= \frac{1}{2J_i+1} \left| (\Psi_f : J_f T_f ||| \hat{\mathcal{O}}_{J,T} ||| \Psi_i J_i T_i) \right|^2 \times \frac{|(T_i T_z, T0 | T_f T_z)|^2}{2T_f+1}. \tag{5.19}$$

Note the last line uses the result of Eq. (5.18).

5.1.5 Sample case: spin-flip

Let's consider a couple of simple cases, both in the *sd* shell with the USDB interaction [Brown and Richter, 2006]. 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 vector of Pauli matrices $\vec{\sigma}$ carries one unit of angular momentum and no isospin, we only use the (`Tt= 0`) set of matrix elements (column second from the right). `BIGSTICK` also generates the transition matrix elements for `Jt =` 2, 3, and 4, not shown. Applying (5.19), 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 $\mathrm{B}(\sigma : 1 \to 25) = 0.3597$.

5.1.6 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

$$ {}^{A}_{Z}X_N \to {}^{A}_{Z\pm1}\, Y_{N\mp1}, $$

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:

$$ B(\mathcal{O} : i \to f) = \frac{1}{2J_i+1} \left| (\Psi_f : J_f || \hat{\mathcal{O}}_J || \Psi_i J_i) \right|^2 $$

$$ = \frac{1}{2J_i+1} \left| (\Psi_f : J_f T_f ||| \hat{\mathcal{O}}_{J,T} ||| \Psi_i J_i T_i) \right|^2 \times \frac{|(T_i T_{z,i}, T \pm 1 | T_f T_{z,f})|^2}{2T_f+1}, \quad (5.20) $$

where the difference between Eq. (5.19) and (5.20) is in the isospin Clebsch-Gordan. There is, however, one more subtle point in treating the isospin raising/lowering operator, $\tau_\pm$. If one treats τ as a rank-1 spherical tensor in isospin space, one can show that

$$ \tau_\pm = \frac{1}{\sqrt{2}} \tau_{1,\pm1}. \quad (5.21) $$

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

$$ \sum_f B(\vec{\sigma}\tau_+ : i \to f) - B(\vec{\sigma}\tau_- : i \to f) = 3(N-Z) \quad (5.22) $$

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.

5.1.7 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 `Tt= 1` one-body density matrix elements, we get B(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(GT) = 0.1654, for either $\beta+$ or $\beta-$.

5.1.8 Transitions utilities

A set of codes for generating common one-body operator matrix elements and for post-processing of one-body density matrices into, e.g., transition B-values, is available to download from GitHub, in the `/util/` directory of `https://github.com /cwjsdsu/BigstickPublick/`. A detailed manual and sample runs are included.

5.2 Two-body densities

`BIGSTICK` can now compute two-body densities. (At this time, two-body densities have not yet been enables for MPI.) The two-body density matrix elements are defined in parallel to one-body densities. We want

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

so we define

$$\begin{aligned} & \rho_J^{fi}(ab, J_{ab}; cd, J_{cd}) \\ \equiv [J]^{-1} & \left\langle J_f \left|\left| \left[\hat{A}^\dagger_{J_{ab}}(ab) \otimes \tilde{A}_{J_{cd}}(cd) \right]_J \right|\right| J_i \right\rangle \frac{1}{\sqrt{(1+\delta_{ab})(1+\delta_{cd})}} \qquad (5.23) \\ = -[J]^{-1} & \left\langle J_f \left|\left| \left[\left[\hat{a}^\dagger \otimes \hat{b}^\dagger \right]_{J_{ab}} \otimes \left[\tilde{c} \otimes \tilde{d} \right]_{J_c d} \right]_J \right|\right| J_i \right\rangle \zeta_{ab}^{-1} \zeta_{cd}^{-1}, \end{aligned}$$

where the factor $\zeta_{ab} \equiv \sqrt{1+\delta_{ab}}$ is needed for normalized two-body states; see Appendix A.2. In proton-neutron format, $\zeta = 1$ always, that is, proton and neutron orbitals are considered distinct.

Note also that we have defined

$$\tilde{A}_{J\,M}(ab) = -\left[\tilde{a} \otimes \tilde{b}\right]_{JM} = (-1)^{J+M} \hat{A}_{J\,-M}(ab). \tag{5.24}$$

where the time-reversed operator is

$$\tilde{c}_{j_c,\,m_c} = (-1)^{j_c+m_c} \hat{c}_{j_c,\,-m_c}. \tag{5.25}$$

so that

$$\hat{A}_{JM}(ab) \equiv -\left[\hat{a} \otimes \hat{b}\right]_{JM} = \left(\hat{A}^{\dagger}_{JM}(ab)\right)^{\dagger} \tag{5.26}$$

$$= -(-1)^{J-M} \left[\tilde{a} \otimes \tilde{b}\right]_{J\,-M}. \tag{5.27}$$

The two-body matrices so defined are reduced with respect to angular momentum but not with respect to isospin.

One must carry out an ordinary run to create a `.wfn` file, e.g., options such as '`n`', '`d`', etc.. Then run `BIGSTICK` again, choosing '`2`' on the initial menu. You will be asked for the name of the previously computed `.wfn` file, as well as the mandatory name of the output file. The resulting file will be have a extension `.den2b`. You will also be asked

```
Enter start, stop for initial states
(This is because two-body densities are large)
(Enter 0,0  to read all )
```

with a similar choice for final states.

The output `.den2b` file begins by defining the proton and neutron orbits:

```
!#  Two-body densities from BIGSTICK version 7.9.8  Sept  2020
!#  Run date: 2020-02-15
!# Densities written in explicit proton-neutron formalism
!# Single-particle orbits information follows
!# Number of single-particle orbits (same for both protons, neutrons )
          3
!# Proton, neutron index  N     L   2xJ
        1    4           0      2      3
        2    5           0      2      5
        3    6           1      0      1
!#   Z   N
     4   4
```

In this explicit proton-neutron format, orbits 1-3 refers to protons, and 4-6 to neutrons. Then for each set of initial and final states, we get the two-body densities:

```
 !# Ini state      Energy         J
         1       -92.77905       0.0
 !# Fin state      Energy         J
         1       -92.77905       0.0
 !# a   b    Jab    c   d    Jcd  Jmin Jmax   &
  (< f || [[ab:Jab]^+[cd:Jcd]]_J || i >,/sqrt(2J+1) J=Jmin,Jmax)
    1   1     0     1   1     0     0   0       0.199810
    1   1     2     1   1     2     0   0       0.057047
    1   1     2     2   1     2     0   0       0.027265
    1   1     0     2   2     0     0   0       0.428295
    ...
!# Ini state      Energy         J
         2       -91.11964       2.0
 !# Fin state      Energy         J
         2       -91.11964       2.0
 !# a   b    Jab    c   d    Jcd  Jmin Jmax  &
  (< f || [[ab:Jab]^+[cd:Jcd]]_J || i >, J=Jmin,Jmax)
    1   1     0     1   1     0     0   0       0.367200
    1   1     0     1   1     2     2   2       0.024823
    1   1     2     1   1     0     2   2       0.024823
    1   1     2     1   1     2     0   4       0.121627  -999.000000     0.009363  .....
    1   1     0     2   1     2     2   2       0.064846
    1   1     0     2   1     4     4   4      -0.023200
    1   1     2     2   1     1     1   3    -999.000000    -0.018076  -999.000000
    1   1     2     2   1     2     0   4       0.061892  -999.000000    -0.012186  .....
    1   1     2     2   1     3     1   4    -999.000000    -0.021697  -999.000000   ....
    1   1     2     2   1     4     2   4       0.036903  -999.000000     0.000209
    1   1     0     2   2     0     0   0       0.785513
    1   1     0     2   2     2     2   2       0.084297
...
```

What this means is as follows: we couple up two destruction operators, with orbit labels c and d to total angular momentum `Jcd`, and two creation operators in orbits a and b coupled up to `Jab`, and the total transition operator is coupled up to some J. The range of J is from `Jmin = |Jab-Jcd|` to `Jmax = Jab+Jcd`; furthermore, we must have $|J_i - J_f| \leq J \leq J_i + J_f$. For each allowed value of J, we have a value of the density matrix.

A value of `-999.0000` means that the matrix element could not be computed due to a vanishing Clebsch-Gordan coefficient; in that case, one should rerun with a different M value.

'Diagonal' two-body densities

An alternate option is to compute only the "diagonal" densities, which we define here as the same initial and final states and only scalar densities, that is, only $J = 0$ in the output. While the usual two-body densities in proton-neutron formalism are still written to a `.den2b` file, one can reinterpret these as expectation

values of operators, which have a different normalization, written to the `.res` file. The options are '`2d`,' which writes the expectation values in proton-neutron formalism, and '`2i`', which write the expectation values in isospin formalism.

With these restrictions, less information is needed:

```
!#   State       Energy          Jstate
        1        -40.47233        0.0
!# a   b   c   d   Jpair   <  psi || [[ab:Jpair]^+[cd:Jpair]]_0 || psi >
   1   1   1   1     0       0.073358
   1   1   1   1     2       0.011630
   1   1   2   1     2       0.011295
   1   1   2   2     0       0.200507
   1   1   2   2     2       0.035984
   1   1   3   1     2       0.015691
```

Note that `Jpair` here is the angular momentum of the pair. Because we have time-reversal symmetry, we have that $\rho(ab,J;cd,J) = \rho(cd,J;ab;J)$. For the diagonal case, only one value is printed.

From this one can extract the expectation value of components of the Hamiltonian. Let

$$\hat{O}_J(ab,cd) = \zeta_{ab}^{-1}\zeta_{cd}^{1}\sum_M \hat{A}^\dagger_{JM}(ab)\hat{A}_{JM}(cd) \tag{5.28}$$

a scalar operator. Now define for a state i the expectation value

$$\langle i, J_i M|\hat{O}_J(ab,cd)|i,J_iM\rangle = \frac{[J]}{[J_i]}\rho_0^{ii}(ab,J;cd,J). \tag{5.29}$$

Note that, because of time-reversal symmetry, $\langle i, J_i M|\hat{O}_J(ab,cd)|i,J_iM\rangle = \langle i, J_i M|\hat{O}_J(cd,ab)|i,J_iM\rangle$. In a Hamiltonian, which is time-reversal-symmetric, a matrix element $V_J(ab,cd)$ actually applies to both. Therefore we define an expectation value which is the sum of both,

$$\begin{aligned} X^i_J(ab,cd) &\equiv \frac{\Big(\langle i, J_i M|\hat{O}_J(ab,cd)|i,J_iM\rangle + \langle i, J_i M|\hat{O}_J(cd,ab)|i,J_iM\rangle\Big)}{(1+\delta_{ac}\delta_{bd})} \\ &= \frac{2}{1+\delta_{ac}\delta_{bd}}\frac{[J]}{[J_i]}\rho_0^{ii}(ab,J;cd,J) \end{aligned} \tag{5.30}$$

This is equivalent to taking the expectation value of a Hamiltonian with exactly one matrix element, $V_J(ab,cd) = 1$. It is important to note that proton and neutron orbits are defined to be different, so that $a_p \neq a_n$. Such expectation values are useful for, e.g., perturbative fitting of matrix elements. It is also useful as a consistency check, as one should get the sum rule

$$\begin{aligned} \sum_{ab}\frac{1+\delta_{ab}}{2}X^i_J(ab,ab) &= \sum_{a\leq b}\sum_J X^i_J(ab,ab) \\ &= 2\sum_{a\leq b}\sum_J \frac{[J]}{[J_i]}\rho_0^{ii}(ab,J;ab,J) = A(A-1), \end{aligned} \tag{5.31}$$

where A is the number of valence particles. When running this option, `BIGSTICK` automatically prints out this sum rule.

If one choose '`(2d)`' the expectation values are printed in proton-neutron formalism. One can also convert the densities to isospin format, using option '`(2i)`', though be aware, if the state has nonzero isospin, then one can have isospin tensors, i.e., non-isoscalar operators. Nonetheless we can generalize (5.28)

$$\hat{O}_{JT}(ab,cd) = \frac{\zeta_{ab}^{-1}\zeta_{cd}^{-1}}{1+\delta_{ac}\delta_{bd}} \sum_{M,M_T} \hat{A}^\dagger_{JM,TM_T}(ab)\hat{A}_{JM,TM_T}(cd) + \hat{A}^\dagger_{JM,TM_T}(cd)\hat{A}_{JM,TM_T}(ab), \tag{5.32}$$

a scalar, isoscalar, time-symmetric operator, and extract expectation values

$$\begin{aligned} X^i_{J,T=1}(ab,cd) &\equiv \langle i, J_iM, T_iM_T|\hat{O}_{JT=1}(ab,cd)|i, J_iM, T_iM_T\rangle \\ &= X^i_J(a_pb_p,c_pd_p) + X^i_J(a_nb_n,c_nd_n) + \frac{1}{2\sqrt{(1+\delta_{ab})(1+\delta_{cd})}} \\ &\times \big[X^i_J(a_pb_n,c_pd_n) + (-1)^{j_a+j_b+j_c+j_d}X^i_J(b_pa_n,d_pc_n) \\ &-(-1)^{J+j_a+j_b}X^i_J(b_pa_n,c_pd_n) - (-1)^{J+j_c+j_d}X^i_J(a_pb_n,d_pc_n)\big] \end{aligned} \tag{5.33}$$

and

$$\begin{aligned} X^i_{J,T=0}(ab,cd) &\equiv \langle i, J_iM, T_iM_T|\hat{O}_{JT=0}(ab,cd)|i, J_iM, T_iM_T\rangle \\ &= \frac{1}{2\sqrt{(1+\delta_{ab})(1+\delta_{cd})}} \\ &\times \big[X^i_J(a_pb_n,c_pd_n) + (-1)^{j_a+j_b+j_c+j_d}X^i_J(b_pa_n,d_pc_n) \\ &+(-1)^{J+j_a+j_b}X^i_J(b_pa_n,c_pd_n) + (-1)^{J+j_c+j_d}X^i_J(a_pb_n,d_pc_n)\big] \end{aligned} \tag{5.34}$$

Note in the mixed proton-neutron contributions, we keep fixed the order pn, pn, as that is how `BIGSTICK` orders the labels. In the above, the labels a, b, etc. without suffixes are shared by both protons and neutrons.

It is important to remember: These expectation values are written to the output `.res` file, with the reguarly-defined densities still written to the `.den2b` file. Furthermore, the 'densities' and the 'expectation values' have different normalizations.

If the state has isospin $T_i > 0$, however, there are expectation values for up to four additional non-isoscalar operators (one isotensor and three isovector). We leave those as an amusing exercise for the reader.

5.3 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, or option '`s`' in the main menu. Note that this default strength function option, as well as '`sn`' , will write *normalized* wave functions to file. The option '`su`' will write unnormalized wave functions to file. This is useful in some important applications, as discussed below in 5.3.3 and 5.3.4.

5.3.1 Decomposition

We'll start with decomposition of a wavefunction using a scalar operator [Johnson, 2015], because operationally it is the most straightfoward. 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 $\hat{\mathcal{O}}$ 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 $\hat{\mathcal{O}}$ in turn has eigenpairs,

$$\hat{\mathcal{O}}|\Phi_\omega\rangle = \omega|\Phi_\omega\rangle. \tag{5.35}$$

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

$$|\Psi\rangle = \sum_\omega c_\omega|\Phi_\omega\rangle \tag{5.36}$$

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 $\hat{\mathcal{O}}$ 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 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 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 labled 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
```

Often the norm or total input strength is far different from 1.

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.

5.3.2 Transition strength function distributions: the basics

Often we want the transition function between two states, that is $\left|\langle\Psi_f|\,\hat{\mathcal{O}}\,|\Psi_i\rangle\right|^2$ where $\hat{\mathcal{O}}$ is some one-body transition operator, for example the $E2$ or $M1$ tran-

sition 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 5.1.4.

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, $\hat{\mathcal{O}}$. 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 ||| O ||| 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.

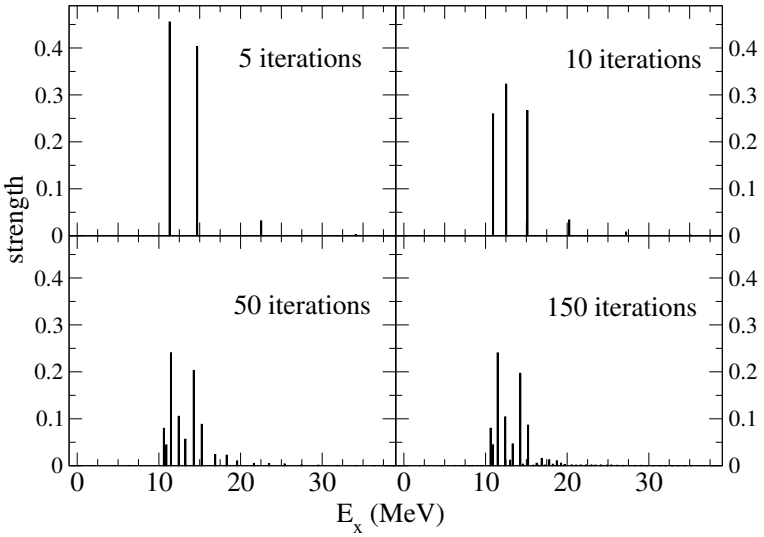

Figure 5.1: Illustration of how transition strengths evolve with increasing number of Lanczos iterations. In this example, the operator $\vec{\sigma}\tau_0$ was applied to the ground state of ^{28}Si, calculated in the *sd* shell with the USDB interaction.

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.

This can be confusing at first, as illustrated in Fig. 5.1. Here we computed in the *sd* shell and using the USDB interaction [Brown and Richter, 2006]m ^{28}Si (which has six valence protons and six valence neutrons on top of a frozen ^{16}O core). After generating the wave function in a 'normal' run (option '`(n)`', we applied the operator $\vec{\sigma}\tau_0$ (generated with additional tools now available in the `BIGSTICK` GitHub repository) using the option '`(o)`'. Finally, we used the strength function option '`(s)`', and applied 5, 10, 50, and 150 iterations. When the individual strengths are plotted, it looks like the results are badly converged.

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 Caurier et al. [2005] and references therein, 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.

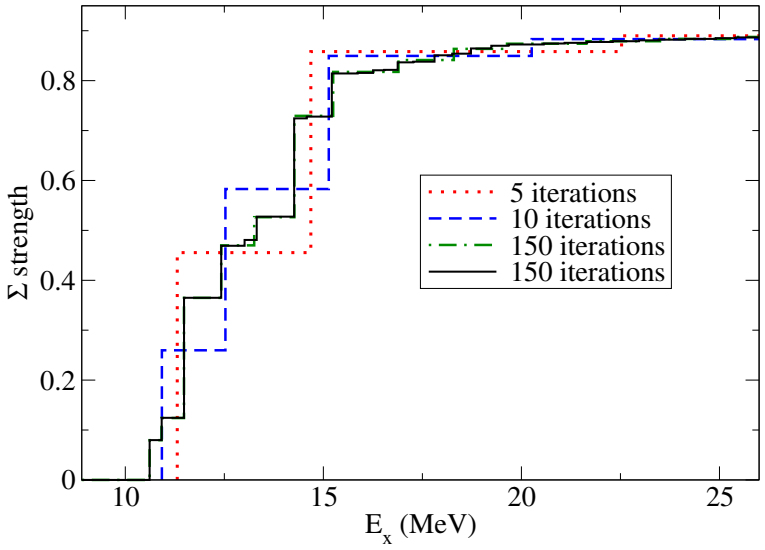

Figure 5.2: Running sums of the strengths shown in Fig. 5.1.

Hence often only thirty or fifty iterations suffice. This is illustrated in Fig. 5.2, which displays the running sums of the strengths in Fig. 5.1. Plotted thusly, one can see that how well they overlap, and indeed the 50 iteration case is nearly indistinguishable from the 150 iteration case.

As another 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.

Note: the option '`ss`' will only compute the strength distribution, but will not write wave functions to file, nor compute J, T of the final wave functions. This is useful for large dimension cases with many iterations, as the wavefunctions are often not used afterwards.

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.

5.3.3 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. (5.17) 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 and 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:

$$B(\mathcal{O}:i\to f)=\frac{1}{2J_i+1}\left|(\Psi_f:J_f||\hat{\mathcal{O}}_J||\Psi_iJ_i)\right|^2= \frac{2J_f+1}{(2J_i+1)}\left|\frac{\langle\Psi_f:J_fM|\hat{\mathcal{O}}_{J0}|\Psi_i:J_iM\rangle}{(J_iM,J0|J_fM)}\right|^2. \tag{5.37}$$

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

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

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`,

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

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 5.3.1.

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. (5.37), 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. (An efficient way to do this is using the block strength function option.)

(Note: these examples need to be reworked using normalization forwarding in 7.9.8).

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, roundoff error allows them to grow during Lanczos. This is the same phenomenon which forces us to orthogonalize new Lanczos vectors against old ones.) 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. (5.37):

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

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.

5.3.4 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 $T_{z,0} = \min(\mathrm{abs}(T_{z,i}), \mathrm{abs}(T_{z,f}))$ and invoke isospin rotation. Typically you will have to filter both J and T. The B-value is given by

$$\begin{aligned} B(\mathcal{O}: i \to f) = \frac{1}{2J_i+1} \left|\langle \Psi_f : J_f, T_f\, T_{z,f}||\hat{\mathcal{O}}_J||\Psi_i : J_i, T_i\, T_{z,i}\rangle\right|^2 = \\ \frac{2J_f+1}{(2J_i+1)} \left| \frac{\langle \Psi_f : J_f M, T_f\, T_{z,0}|\hat{\mathcal{O}}_{1\,0,1\,0}|\Psi_i : J_i M, T_i\, T_{z,0}\rangle}{(J_i M, J0|J_f M)} \right|^2 \\ \times \left| \frac{(T_i\, T_{z,i}, 1\ \pm 1|T_f\, T_{z,f})}{(T_i\, T_{z,0}, 1\,0|T_f\, T_{z,0})} \right|^2 \end{aligned} \tag{5.38}$$

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 $|\langle \Psi_f : J_f M, T_f\, T_{z,0}|\hat{\mathcal{O}}_{1\,0,1\,0}|\Psi_i : J_i M, T_i\, T_{z,0}\rangle|^2$. 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:

```
 1.0000000877719881       = 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 $J;T = 5/2;1/2 \to 3/2;1/2$ 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. (5.38):

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

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.

5.4 Resolvent/Green's function

One can compute the action of the Green's function or resolvent, $(E_0 - \hat{H})^{-1}$ on an initial state, the pivot $|\Psi_i\rangle$, which must be read in from a file. The option `'(g)'` writes $(E_0 - \hat{H})^{-1}|\Psi_i\rangle$ to a `.wfn` file.

In the related option, `'(gv)'`, after calculating the resolvent on the pivot, one reads in a previously computed file of wave functions and computes the overlap.

After reading in the interaction, you will be asked for the energy

```
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
+ Calculating resolvent/Green function on an initial vector


Enter energy E in resolvent/Green function 1/(E-H)
```

If E_0 is extremal relative to the eigenvalues, Lanczos algorithm converges quickly, but if it is interior, the convergence can be slow. This can be understood through a spectral decomposition of the uncoverged eigenvalues, crossing E_0.

The option `'(gc)'` allows you to enter a complex energy E_0.

```
Enter real, imaginary parts of E
```

The output is written to a file, as two vectors: the first vector is the real part, the second vector is the imaginary part. You can then take overlaps or use as necessary.

The most general option is `'(pv)'`. This option reads in a choice of pivot and carries out a fixed, user-defined number of iterations. The resulting Lanczos vectors are then dotted against a second `.wfn` input file, with the results written to the file `overlap_lanczos.dat`:

```
 Lanczvec     Finalstate       overlap
       1         1         -0.01173
       2         1          0.04016
       3         1         -0.09216
       4         1          0.15468
. . . .
```

No energy is input, but from the Lanczos coefficients and overlaps with the Lanczos vectors, one can reconstruct the resolvent for an arbitrary energy.

Chapter 6

A peek behind the curtain

Although this manual details how to use `BIGSTICK`, it only outlines the algorithms and program. 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 program, with more than seventy Fortran files and on the order of 70,000 lines. Several files are for specialized applications most users will not care about, as well as for features slated for obsolescence.

There are ways to get `BIGSTICK` to present more information about its inner workings, as well as ways to extert 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 Writing out 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.

To facilitate user-written post-processing, we have added two additional options. Option `'(tx)'` allows you to read in a standard format .wfn file and to write it out in the .trwfn format. Option `'(tw)'` goes in the opposite direction: with it one can read in a .trwfn file and write out as a .wfn file. For this latter option, the user **must** enter in the information about the basis as for a normal `BIGSTICK` run. Furthermore, the order of the coefficients in the .trwfn file cannot be changed; when reading in from a .trwfn file, `BIGSTICK` assumes the order of the coefficients is standardized and does not check the actual occupations of the configurations.

6.2.1 Alternate information on basis

To enable further post-processing, two new options for printing out basis information have been added. Option `'(b)'` will produce a binary file, with extension `.bas`, with basis information. Alternately, and likely more useful for most readers, option `'(ba)'` will produce a human-readable (ASCII) version of the same information, also with extension `.bas`. Unlike the option `'(t) ,'` this option does not create the full basis. It does provide valuable information, however, on how the full basis is constructed from proton and neutron many-body states (Slater determinants–technically, the occupation representation of Slater determinants–or 'SDs'). The basic idea is that each proton SD and each neutron SD is assigned an index, `ip` and `jn`, respectively, and there are arrays `pstart()` and `nstart` which provide the index of the combined many-body state, that is,

`index = pstart(ip) + nstart(jn).`

Furthermore, the occupied single particle states for each proton and neutron SD are provided. Hence the `.bas` provides information on how to reconstruct the basis.

After writing the basis information to file, `BIGSTICK` halts. These options do not create or solve the Hamiltonian matrix, and do not write any wave function vectors to file; for that you will still need the `(t)` option. The many-body states and amplitudes written to the `.trwfn` file will be in the same order as those defined

(The example given is for ^{20}Ne in the *sd*-shell. Some of the tabs may be different in your output file. In addition, the comments do not appear in the actual `.bas` file.)

```
    31415926        ! 'magic number'
                  1         ! wfn version number
```

The 'magic number' was introduced in case we made significant changes to our wave function conventions. The version number serves a similar function

```
  2      2     ! valence Z and N
 T    ! isospin flag (hardly used any more)
  3       3 ! # of proton, neutron orbits
```

The quantum numbers of the orbits (protons first, then neutrons, even if the same) are, in order: nr (radial quantum number), $2 \times j$, l, π (parity) and w (weighting for truncations):

```
!  nr   2j     l   par  w
    0    3    2    1    0
    0    5    2    1    0
    1    1    0    1    0
    0    3    2    1    0
    0    5    2    1    0
    1    1    0    1    0
   12     12  ! # of proton, neutron single particle states
```

The quantum numbers of the single particle states (protons first, then neutrons, even if the same), are, in order: nr (radial quantum number), $2 \times j$, $2 \times m$, l, w (weighting for truncations), π (parity), orbit label corresponding to the above defined orbits, and finally the group number. Here the 'group' is used to identify single particle states with the same parity, w, and m value, used in creating arrays for interactions; however it is not set by the time the basis is written to file (and is not generally needed).

```
!  nr   2j    m   l     w  par orb  group
    0    3   -1    2    0    1    1    0
    0    3    1    2    0    1    1    0
    0    5   -1    2    0    1    2    0
    0    5    1    2    0    1    2    0
    1    1   -1    0    0    1    3    0
    1    1    1    0    0    1    3    0
```

```
    0    3    3    2    0    1    1    0
    0    5    3    2    0    1    2    0
    0    5   -3    2    0    1    2    0
    0    3   -3    2    0    1    1    0
    0    5   -5    2    0    1    2    0
    0    5    5    2    0    1    2    0
...
```

Because this information is mostly repeated later: `BIGSTICK` initially reads in or creates a single-particle space, and later, depending upon truncations, may only use part of that single particle space. This is mostly in the context of *ab initio* / 'no-core' calculations. For example, one may define a single-particle space with 10 major oscillator shells, but then create a p-shell nucleus allowing only $4\hbar\Omega$ excitations, equivalent to choosing `Max excite` (W) of 4; this will only use 6 major oscillator shells. The reason for this is for the code to be more robust to use; also, for some interaction files the single-particle space in the file may be larger than that needed, in which case one must initially put in the single-particle space appropriate for the interaction file. If you are doing phenomenological calculations in a restricted space, this is generally not relevant for you.

The following quantum numbers are for the full many-body space:

```
          0 +       0    ! Jz   parity  max W
```

The following are logical flags for single-particle space:

```
 T T F   ! allsameparity, allsameW, spinless
             640  ! basis dimension
```

If there are no truncations, for example as in many phenomenological calculations, `allsameW=.true.` and all values of w are set $=0$.

Now the single-particle states are written out again. As stated above, it may seem redundant, but it ensures that the many-body states are correctly interpreted. The following single-particle states are the minimum needed. For phenomenological calculations, such as in the sd or pf shell, this is usually the same information as before.

```
          6        6       0       6       6    ! nhsps(-2:2)
```

These are dimensions of subspaces of the single-particle states. The number of proton single particle states used is `nhsps(1)+nhsps(-1)`; the number of neutron single particle states used is `nhsps(2)+nhsps(-2)`; `nhsps(0)` contains no useful information. This information is given above but is repeated as a check

Next come (again) the single particle quantum numbers, here the single particle quantum states actually used. These are needed to interpret the many-body states

```
! index   nr   l   2j   2m    w  par
     1    0    2    3   -1    0    1
     2    0    2    5   -1    0    1
     3    1    0    1   -1    0    1
     4    0    2    5   -3    0    1
     5    0    2    3   -3    0    1
     6    0    2    5   -5    0    1
     7    0    2    3    1    0    1
     8    0    2    5    1    0    1
...
```

Finally the many body states are written out, proton Slater determinants first, then neutron. It is important to understand that the basis is organized by sectors. A sector is the set of proton or neutron Slater determinants defined by quantum numbers J_z, π, and W.

```
           9        66   ! # of proton sectors, proton Slater dets
```

That is, in this example, there are 9 proton sectors, with a total of 66 proton Slater determinants distributed among them.

Looping over the proton sectors,

```
     1      -8      1       0  ! index, 2Jz, par, and W for sector
     1       2      2   ! start_pSD, stop_pSD,  # proton SDs in sector
```

One should have `stop_pSD - start_pSD + 1` = number of proton Slater determinants in this sector.

Each proton sector has a *conjugate* neutron sector; their quantum numbers J_z, π, W combine correctly to the quantum numbers for the full system, i.e., $J_z(p) + J_z(n) = J_z(\mathrm{tot})$.

```
           1           ! # of conjugate sectors for this sector
           1           ! list of conjugate neutron sectors
```

One only gets more than one conjugate sector when doing non-trivial truncations in W.

```
           0           0    !   obsolete information
```

Finally, for each proton sector, the proton Slater determinants are listed in order, starting from `start_pSD` and ending with `stop_pSD`. They are listed with: label;

```
!   label           pstart       occupied s.p. states...
           1              0           5           6
           2              2           4           6
```

Here `pstart` is a key array; neutron Slater determinants have a similar array `nstart`. When combining a proton Slater determinant and a neutron Slater determinant, the sum `pstart+nstart` is the index in the combined basis.

For each Slater determinant, the list of occupied single particle states (using the second indexing given above) describes the state.

After looping over the proton sectors and listing the occupied single particle states in each proton Slater determinant in the sector, the process starts over again with neutron sectors.

The pseudocode for construction of the basis:

loop over proton sectors
  loop over list of neutron sectors conjugate to current proton sector
    loop over proton Slater determinants `ip` *in proton sector*
      loop over neutron Slater determinants `jn` *in neutron sector*
        basis index = `pstart(ip) + nstart(jn)`

The routines for writing to the binary `.bas` file are `write_wfn_header` in `bwfnlib.f90` and `basis_out4postprocessing` in `boutputlib.f90`; for the human-readable ASCII file `write_wfn_header_ASCII` is used instead.

6.3 Writing out the Hamiltonian and other operators

The many-body Hamiltonian can also be explicitly generated by choosing the option '`(wh)`'. This will write the nonzero matrix elements to the file `ham.dat`.The first line of the file is the basis dimension. The following lines are i, j, H_{ij}, with $i \geq j$, and only for H_{ij} nonzero. Choosing '`wo`' will write out the matrix elements of a one-body transition operator in the same format. After writing the nonzero matrix elements to file, `BIGSTICK` halts and does not solve the eigenvalue problem.

Alternately, 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[Parlett, 1980, Press et al., 1992], 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 would one want to.

(We can understand why through the concept of intruder states, that is, a state outside the designated model space. For example, in the *sd*-shell, one has only positive parity states, so any negative parity state is an intruder. Yet, experimentally, at some point intruders–in our example, negative parity states, consisting of excitation from the p shell into the *sd* shell, or from the *sd* shell into the pf shell–will start appearing in the experimental spectrum, but are outside the calculation. Eventually intruder states dominate, simply because there are an infinite number of them, and any calculation in a finite model space is physically incomplete. 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 [Parlett, 1980, Press et al., 1992, Whitehead et al., 1977]. 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

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

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

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 eventuall 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, eventualy 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 no one clearly successful recipe. `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 jth 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*[Wu and Simon, 2000]. 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 otherwords, 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}, and the consequent number of vectors stored, 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

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

(Actually, with k iterations, although there are $k+1$ Lanczos vectors, the value of α_k hasn't yet been determined. It is not needed, however, at this point, and will be found later.) Suppose, however, we only diagonalize $\mathbf{T}^k$, that is, stopping at the kth column and row, with $\mathbf{L}$ being the $k \times k$ unitary matrix of eigenvectors, that is,

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

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

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

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

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

The key to thick-restart Lanczos is to judiciously truncate this. If you want to get the lowest N_{keep} states, truncate to N_{thick} (with $N_{\text{keep}} < N_{\text{thick}} < k$) vectors, that is, to take from Eq. (7.6) only the first N_{thick} new Lanczos vectors,

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

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

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

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

$$\mathrm{H}|v'_{N_{\rm thick}+1}\rangle = \tag{7.9}$$
$$\beta_k L_{k1}|v'_1\rangle + \beta_k L_{k2}|v'_2\rangle + \ldots + \alpha_{N_{\rm thick}+1}|v'_{N_{\rm thick}+1}\rangle + \beta_{N_{\rm thick}+1}|v'_{N_{\rm thick}+2}\rangle$$

(This, incidentally, is when we find α_{k+1}, only now rebranded as $\alpha_{N_{\rm thick}+1}$.) This first step is not a tridiagonal relation; furthermore , although our new $|v'_{N_{\rm thick}+1}\rangle$ is the same as our old $|v_{k+1}\rangle$, and our new $\alpha_{N_{\rm thick}+1}$ is the same as the old α_{k+1}, the new vector $|v'_{N_{\rm thick}+2}\rangle$ is *not* the same as $|v_{k+2}\rangle$ would have been had we continued the previous iteration, although the former contains the latter as a component, because we orthogonalize $|v'_{N_{\rm thick}+1}\rangle$ against a different set of vectors.

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

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

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 $N_{\rm thick}$ or k, but one should take k as large as practical, and "typical" values of $N_{\rm thick} \approx 3N_{\rm keep}$ 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.

7.2.1 Targeted thick-restart Lanczos: interior eigenvalues

The Lanczos algorithm naturally leads to extremal eigenvalues, but sometimes one wants to obtain interior eigenvalues, that is, highly excited eigenstates. The targeted thick-restarted option, '`(tx)`,' is an approximate method to obtain such interior eigenvalues. The way it does so is by doing thick-started on $(\mathbf{H} - \bar{E})^2$, where $\bar{E}$ is the energy target; then eigenstates near to $\bar{E}$ are low in the synthetic spectrum.

7.3 Block Lanczos

Block Lanczos is a variant where instead of carrying out $\mathbf{H}\vec{v}_i = \vec{v}_f$ on a single vector, one applies the Hamiltonian matrix to a *block* of vectors. We have two motivations for introducing block Lanczos.

The first motivation is the inherent inefficiencies of our factorization/on-the-fly algorithm. We do not store all the nonzero many-body matrix elements, but largely reconstruct them on the fly. This save us tremendously on storage of matrix elements, it takes time to reconstruct the matrix elements. (Numerical experiments suggest roughly a factor of two difference in matrix-multiplication time between storage and on-the-fly, which is surprisingly good.) Furthermore, out of necessity the algorithms for reconstruction proton-proton, neutron-neutron, and proton-neutron matrix elements are different and can take different amounts of time; furthermore those times can have large fluctuations in them, which leads to difficulty in load-balancing.

Furthermore, since the M-scheme matrix is very sparse, the indices of the vector elements accessed can be very far apart, leading to a loss of data locality. The complexity of the on-the-fly reconstruction algorithm only makes this worse.

To be explict, consider the standard matrix-vector multiplication (matvec mult), $\mathbf{H}\vec{v} = \vec{w}$

$$w_i = \sum_j H_{ij} v_j. \tag{7.10}$$

In our standard algorithm, the value of the matrix element H_{ij} as well as the indices i, j are reconstructed on the fly. Only a small number of the H_{ij} are nonzero, hence a very sparse matrix, and approximately, the indices are called randomly.

(In fact, this work can be distributed over many MPI processes, ordered by sectors defined by quantum numbers, and the final index i is ordered to avoid race conditions in OpenMP parallelization, but these details are unimportant to the discussion here.)

In block Lanczos, however, one applies the same equation to multiple vectors:

$$w_{i,a} = \sum_j H_{ij} v_{j,a}, \tag{7.11}$$

where a labels the different vectors in the blocks. Because reconstruction H_{ij} is relatively expensive, as well as variable in the cost in time, block Lanczos

amortizes the cost by applying it to multiple vectors. Furthermore, for the matrix-matrix multiplication we store the vector blocks row-wise rather than column wise. In pseudocode this looks like:

```
loop over matrix elements;
    fetch Hij,i,j
    loop over a
        w(a,i) = w(a,i) + Hij * v(a,j)
    end loop
end loop
```

Because the elements of the block vector `w(a,i)` and `v(a,j)` are contiguous, i.e., *local* in memory with respect to `a`, this dramatically reduces cache calls. The resulting factor of 2 speedup makes this part of the algorithm equal in speed to codes which store the matrix elements explicitly in memory.

The downside of block Lanczos is that it can require significantly more total iterations than standard Lanczos.

Block Lanczos is well described by [Shimizu et al., 2019], who focus on thick-restart block Lanczos; for this option in `BIGSTICK`, see Section 7.3.3.

Running in block Lanczos mode is described above in section 4.6.2. For important constraints when running in parallel using MPI, read section 8.1.2.

Here is how we carry out block Lanczos. Let $N_{\rm dim}$ be the M-scheme dimension, and let $N_{\rm block}$ be the dimensions of the blocks (which is `dimblock` in the code), that is, the number of vectors in each block. Then let $\mathbf{V}_n$ be the nth block of vectors, it is a $N_{\rm dim} \times N_{\rm block}$ rectangular matrix. The column vectors of $\mathbf{V}_n$ should be orthonormal, not only to each other but also to all column vectors in all other block $\mathbf{V}_m$.

The block $\mathbf{V}_n$ are generated iteratively by matrix multiplication, as in the standard (or vector) Lanczos. When carrying out the matrix multiplication, however, the transpose is stored, so as improve locality, that is, although we write

$$\mathbf{H}\,\mathbf{V}_n = \mathbf{W}_n, \tag{7.12}$$

where $\mathbf{W}_n$ is a temporary matrix, we actually do

$$\mathbf{V_n}^T\mathbf{H} = \mathbf{W}_n^T,$$

to reduce cache calls. In the rest of this discussion, however, we suppress this.

The basic block Lanczos iteration is

$$\mathbf{H}\,\mathbf{V}_n = \mathbf{V}_{n-1}\mathbf{B}_{n-1} + \mathbf{V}_n\mathbf{A}_n + \mathbf{V}_{n+1}\mathbf{B}_n, \tag{7.13}$$

where $\mathbf{A}_n, \mathbf{B}_n$ are $N_{\rm block} \times N_{\rm block}$ square matrices; the $\mathbf{A}$ matrices are symmetric, but not the $\mathbf{B}$ matrices. To remove the first term, we orthogonalize $\mathbf{W}_n$ against all previous blocks. Then we compute

$$\mathbf{V}_n^T\mathbf{W}_n = \mathbf{A}_n, \tag{7.14}$$

and then subtract

$$\mathbf{W}_n - \mathbf{V}_n\mathbf{A}_n = \mathbf{W}'_n = \mathbf{V}_{n+1}\mathbf{B}_n. \tag{7.15}$$

To extract $\mathbf{V}_{n+1}$ and $\mathbf{B}_n$, compute the $N_{\text{block}} \times N_{\text{block}}$ symmetric overlap matrix

$$\mathbf{O} = \mathbf{W}'^T_n \mathbf{W}'_n, \tag{7.16}$$

but this is equal to

$$= \mathbf{B}^T_n \mathbf{V}^T_{n+1} \mathbf{V}_{n+1} \mathbf{B}_n = \mathbf{B}^T_n \mathbf{B}_n \tag{7.17}$$

because of the orthonormality of the column vectors of the $\mathbf{V}$ blocks.

Now we have to factor the overlap matrix. There are multiple options, but we choose to do a spectral decomposition (i.e., diagonalization); in all conceivable cases N_{block} will be relatively small and diagaonalization quick, and by examining the eigenvalues one can easily find and remove singular or near-singular values. (Note: as of this writing, version 7.9.8, that is not yet implemented in block Lanczos.)

Let $\mathbf{U}$ be the unitary matrix formed by the eigenvector of $\mathbf{O}$, which must have positive eigenvalues represented by the diagonal matrix $\mathbf{\Lambda}$. Then $\mathbf{O} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$. From this we can conclude that

$$\mathbf{B}_n = \sqrt{\mathbf{\Lambda}}\mathbf{U}^T \tag{7.18}$$

(the square root is easy to carry out, because $\mathbf{\Lambda}$ is diagonal with positive-definite elements). Finally

$$\mathbf{V}_{n+1} = \mathbf{W}'_n \mathbf{B}^{-1}_n = \mathbf{W}'_n \mathbf{U}\mathbf{\Lambda}^{-1/2}, \tag{7.19}$$

where, again, the inverse is easy to carry out. If one of the elements of $\mathbf{\Lambda}$ is near zero, then it is nearly singular, and one needs to 'restart' with a new vector. This will be implemented at a future date.

We note that the $\mathbf{B}_n$ form the block-sub-diagonal, because $\mathbf{B}_n = \mathbf{V}^T_{n+1}\mathbf{H}\mathbf{V}_n$, that is, in the $n+1$th block-row and the nth column-block.

Now the truncated Hamiltonian in the block-Lanczos representation is

$$\mathbf{T} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{B}^T_1 & 0 & 0 & 0 & \dots \\ \mathbf{B}_1 & \mathbf{A}_2 & \mathbf{B}^T_2 & 0 & 0 & \\ 0 & \mathbf{B}_2 & \mathbf{A}_3 & \mathbf{B}^T_3 & 0 & \\ 0 & 0 & \mathbf{B}_3 & \mathbf{A}_4 & \mathbf{B}^T_4 & \\ \vdots & & & & & \ddots \end{pmatrix} \tag{7.20}$$

which is solved in the usual fashion.

7.3.1 Bootstrapped block Lanczos

Although the matrix-matrix multiplication for block Lanczos is much more efficient than matrix-vector multiplication for vector Lanczos, experience shows that one often needs more Lanczos iterations. One can accelerate block Lanczos by starting from vectors that approximate the final vectors Zbikowski and Johnson [2023]. In order to do this, one must use option '`(np)`' in the initial menu.

7.3.2 Block strength function

An obvious useful application is a block strength function, that is, carrying out the strength function option but for a block of vectors. The menu option for this is '`(sb)`'.

We read in a block of pivot vectors, $|w_1\rangle, |w_2\rangle, |w_3\rangle, |w_4\rangle \ldots$ and we want the strength against (approximate) eigenvectors $|E_r\rangle$, that is, we want

$$|\langle w_i|E_r\rangle|^2 .$$

These initial vectors might not be orthonormal, for example after applying a one-body operator (option '`(o)`'). After being read in, they must be orthonormalized. Let $\mathbf{W_0}$ be the initial block of vectors. As above, form the overlap matrix $\mathbf{O} = \mathbf{W}_0^T\mathbf{W}_0 = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$. Then

$$\mathbf{W}_0\mathbf{U}\mathbf{\Lambda}^{-1/2} = \mathbf{V}_1 \tag{7.21}$$

is now an orthonormal block, and the initial pivot block. Let's think what this means. The orthonormalized vectors, which form the pivot block for the Krylov space, are $|v_1\rangle, |v_2\rangle, |v_3\rangle, |v_4\rangle \ldots$ related by

$$|v_j\rangle = \sum_i |w_i\rangle U_{ij}\lambda_j^{-1/2}, \tag{7.22}$$

where λ_j is the jth eigenvalue of the overlap matrix.

After a set number of iterations, we solve as before

$$\mathbf{T} = \mathbf{L}\tilde{\mathbf{E}}\mathbf{L}^T, \tag{7.23}$$

where $\tilde{\mathbf{E}}$ is a diagonal containing the approximate eigenenergies of $\mathbf{T}$, and $\mathbf{L}$ are the eigenvectors represented in the Krylov space. Specifically, $L_{j,r}$ is the jth Krylov coefficient of the rth eigenvector, that is,

$$|E_r\rangle = \sum_j |v_j\rangle L_{j,r}.$$

By inverting (7.22), that is,

$$|w_i\rangle = \sum_j U_{ij}\lambda_j^{1/2}|v_j\rangle,$$

then

$$\langle w_i|E_r\rangle = \sum_j U_{ij}\lambda_j^{1/2}\langle v_i|E_r\rangle = \sum_j U_{ij}\lambda_j^{1/2}L_{j,r} \tag{7.24}$$

This we can do from the original, let $\mathbf{B}_0 = \mathbf{\Lambda}^{1/2}\mathbf{U}^T$, and then

$$\langle w_i|E_r\rangle = \sum_j (B_0)_{ji}L_{j,r}$$

that is, we get the overlaps from $\mathbf{B}_0^T\mathbf{L}$.

Note: One can also choose '`(sbs)`', in which case the J, T of the final states are not computed and the wave functions are not written to file. This can be a smart choice when carrying out a large large with many iterations.

7.3.3 Thick-restart block Lanczos

The thick-restart block Lanczos method (TRBL) combines the strengths and weaknesses of both the thick-restart and block Lanczos methods. TRBL has been shown to be an effective eigensolver for large-scale shell-model calculations where one desires large numbers of eigenstates [Shimizu, 2013]. The power of the thick-restart block Lanczos method is three-fold. First, time in matrix-matrix (or matrix-block vector) multiplication is reduced due to improved data locality. Matrix-block vector multiplication also amortizes the cost of the on-the-fly matrix element reconstruction algorithm. Lastly, by restarting the block Lanczos process, reorthogonalization time is reduced by restricting the number of Lanczos vectors stored in memory.

In substance, TRBL follows the same iterative process of the conventional block Lanczos method, constructing a block tri-diagonal approximation of the Hamiltonian and computing blocks of Lanczos vectors up to some chosen maximum number of block iterations n_s, resulting in k total Lanczos vectors saved before block Lanczos is restarted. N_{thick} total eigenvectors or n_{thick} block Lanczos iterations worth of approximate eigenvectors are constructed and saved for restart. By construction, one takes a linear combination of Lanczos vectors weighted by the components of the eigenvectors of the truncated space. Lanczos vectors computed at the final block iteration before n_s are used as the initial pivot after restarting, analogous the single-vector thick-restart process. Eigenpairs of the reduced Hamiltonian are used to construct a restarted block Lanczos matrix, Eq. (7.25).

No longer purely block tri-diagonal, the matrix $\mathbf{T}$ after restarting is

$$\mathbf{T} = \begin{pmatrix} \mathbf{E}_{n_{thick}} & \mathbf{r}^T & 0 & 0 & 0 & \dots \\ \mathbf{r} & \mathbf{A}_1 & \mathbf{B}_1^T & 0 & 0 & \\ 0 & \mathbf{B}_1 & \mathbf{A}_2 & \mathbf{B}_2^T & 0 & \\ 0 & 0 & \mathbf{B}_2 & \mathbf{A}_3 & \mathbf{B}_3^T & \\ \vdots & & & & & \ddots \end{pmatrix} \tag{7.25}$$

where $\mathbf{E}_{n_{thick}}$ is a diagonal matrix containing the first ordered N_{thick} eigenvalues of the reduced Hamiltonian. The sub-matrix $\mathbf{r}$ is

$$\mathbf{r} := \mathbf{B}_n \mathbf{U}_{k-N_{block}+1:k,1:n_{thick}}. \tag{7.26}$$

The matrix $\mathbf{U}$ contains the approximate k eigenvectors of the Hamiltonian computed in the reduced space prior to restarting. TRBL works well in situations when one desires many eigenpairs, and one can achieve reasonable convergence restarting block Lanczos with approximately N_{block} eigenvectors or a couple multiples there of. When running the algorithm with large block dimensions, you are generally storing a larger proportion of eigenpairs you are interested in relative to the N_{thick} vectors saved in memory for a restart. One can further accelerate the convergence of the thick-restart block Lanczos method by bootstrapping the pivot, that is, loading in approximate eigenvectors projected in from a different model space as the starting pivot block.

7.4 Can I restart standard Lanczos?

The standard Lanzos 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,the 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 and timing

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-opemp.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 throught the setup in two stages. First, it calculates the number of operations in each matvec, that is, in

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

each update

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

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` attemps 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 =              17 million basis states )
( Default =             500 million basis states )
(  Enter fragment size in millions of states (NEW); enter 0 to use default )
```

Note that the units for the fragment size is **millions** of basis states. If the total basis dimension is less than one million, however, then fragment size is given and read in as the number of basis state.

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 500 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 mulitple 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}.

NB: In versions prior to 7.9.6, the fragment size was given in number of basis states; starting with 7.9.6, it is given in millions of basis states, as described above.

8.1.2 Block Lanczos

Running block Lanczos on large cases on parallel machines using MPI brings additional constraints. For the sake of efficiency, the blocks are actually stored as long vectors. If the vectors are not broken up, then the size of each vector is `dimblock` $\times$ `dimbasis`. If the basis is broken into fragments, the size is `dimblock` $\times$ `dimfragment`. Where a problem can occur is if the dimensions are very large. These blocks-as-vectors are passed via MPI, but standard MPI has a

limit of the size of vectors that can be passed, approximately 2 billion. This limit can be breached if the basis is very large. You should choose a fragment size so that `dimblock` $\times$ `dimfragment` is less than 2 billion. Smaller fragments can sometimes require more MPI processes or ranks, as one should at a minimum have $2 \times$ (`nfragments`)^2 processes.

The code will attempt to warn you about problems, but you will have to read the output as well as the log file to find the warning.

(For people who like to fiddle with code: the relevant subroutines are those such as `br_pull_block1_from_reg` and `br_push_block2_to_reg` in file `bblock_algebra.f90`; there could be others.)

8.1.3 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 betwen 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 optypes 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 calculation 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.

8.1.4 Jump storage and 'greedy' opbundles

BIGSTICK works by factorizing both the basis and the interaction into separate proton and neutron components. For the interaction, the action of operators are stored as 'jumps.' BIGSTICK will boast about its efficiency:

```
total # operations           40912499332 , ~ ops/jump =    163.303177
Effective storage per operation =   0.14767923276870706       bytes
```

for ^{12}C at $N_{\max} = 6$, or

```
total # operations         5140214153296 , ~ ops/jump =    52443.1055
Effective storage per operation =    4.5794092421044884E-004  bytes
```

for ^{60}Zn in the pf shell. Remember that an operation is approximately a matrix element. (Each operation actually represents the action of an operator, and diagonal matrix elements can be the sum of several operations.) You can see that this is very efficient. Model spaces without many-body truncations on W, as in in the second case, are significantly more efficient.

Despite this efficiency, jump storage can become a problem. This is especially true for large no-core shell-model like calculations, and in particular when there is a large difference between N and Z. This can be understood through an apt analogy: imagine a rectangle, with one side representing proton information and the other side neutron information, and the interior representing the combined information. BIGSTICK works by storing the perimeter and not the area. However, this is most efficient when the rectangle is square, that is, has sides of equal length. Long, thin rectangles, conversely, are much less efficient.

When running in MPI mode, BIGSTICK divides up the work by time. The jumps themselves are controlled by opbundles, and the opbundles are split to divide up the work. In very large calculations with large differences between N and Z, however, this can run into a problem, because in order to divide up the word more memory may be required on a particular MPI rank than is available. (This is still less memory than would be required by simple storing the non-zero matrix elements.) This leads to 'greedy' opbundles, which must be handled separately. When modeling or running in MPI, BIGSTICK will provide information about these greedy opbundles in the logfile.

Most users will not need to worry about this. if you do, the main options are:

- Increase the number of MPI ranks;
- Increase the memory available to each MPI rank, for example, by assigning more OpenMP threads. This is advanced parallel work and if you do not know how to do this, you should discuss it with a consultant for your HPC machine;
- You can also change the variable `maxjumpmemory_default` in module `flagger` in the file `bmodules_flags.f90`. It is typically set at either 32 or 64 (Gb). This will depend upon the amount of memory available per MPI rank.

Don't forget you need to set aside memory for the Lanczos vectors as well as storing the uncoupled matrix elements, all of which are also provided when `BIGSTICK` models a run.

- If you make multiple runs that crash or halt, the file `timinginfo.bigstick` may become corrupted. `BIGSTICK` may try to alert you to this fact. Try deleting `timinginfo.bigstick` and re-running.

  Most recently (version 7.9.12) a modified distribution algorithm mostly addresses this. (`BIGSTICK` distributes work by assigning weights to different operations; when faced with a greedy opbundle, the code now simply inflates the weighting by the excess memory requirement. This is not guaranteed to work universally, but has solved several previously intractable cases.)

  And remember–some problems may simply be too large, or too large for the machine available, no matter what.

8.1.5 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 enables, 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 form 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. Some of the work we have carried out is described in Shan et al. [2015, 2017].

8.3 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.

8.3.1 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), reorthognalization, and so on. At the end of a run, BIGSTICK prints out the culmulative 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 decouple matrix elements :   1.999999862164259E-003
Time to compute jumps :   1.899999985471368E-002
Time to compute lanczos :    42.0250000003725
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
```

8.3.2 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.

Chapter 9

Recent additions

In this brief chapter, we list new modifications to be integrated into the rest of the manual.

9.1 Density matrix output

As of 7.9.10, density matrices are now written out exclusive to the `.dres` file. An explicit listing of the single particle orbitals is included at the beginning.

Appendix A

Matrix elements and operators

A.1 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,

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

where $(J_f||\hat{O}_K||J_i)$ is the *reduced matrix element*, which encapuslates the fundamental matrix element independent of orientation, and which in 5.1.4 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 Ms). 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:

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

A.2 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

$$(\hat{\alpha} \times \hat{\beta})_{JM} = \sum_{m_\alpha, m_\beta} (j_\alpha m_\alpha, j_\beta m_\beta | JM) \hat{\alpha}_{j_\alpha m_\alpha} \hat{\beta}_{j_\beta m_\beta}, \tag{A.3}$$

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

$$\hat{A}^\dagger_{JM}(ab) = (\hat{a}^\dagger \times \hat{b}^\dagger)_{JM} \tag{A.4}$$

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

$$\tilde{A}_{JM}(cd) = -(\tilde{c} \times \tilde{d})_{JM} \tag{A.5}$$

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

$$\hat{A}_{JM}(cd) = \left(\hat{A}^\dagger_{JM}(cd) \right)^\dagger = (-1)^{J+M} \tilde{A}_{J,-M}(cd) \tag{A.6}$$

A normalized pair operator is

$$\frac{1}{\sqrt{1+\delta_{ab}}} \hat{A}^\dagger_{JM}(ab) \tag{A.7}$$

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

$$\hat{H} = \sum_{ab} e_{ab} \hat{n}_{ab} + \frac{1}{4} \sum_{abcd} \zeta_{ab} \zeta_{cd} \sum_J V_J(ab, cd) \sum_M \hat{A}^\dagger_{JM}(ab) \hat{A}_{JM}(cd), \tag{A.8}$$

where $\hat{n}_{ab} = \sum_m \hat{a}^\dagger_m \hat{b}_m$ 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

$$\sum_{ab} e_{ab} [j_a] \left(\hat{a}^\dagger \times \tilde{b} \right)_{0,0} + \frac{1}{4} \sum_{abcd} \zeta_{ab} \zeta_{cd} \sum_J V_J(ab, cd) [J] \left(\hat{A}^\dagger_J(ab) \times \tilde{A}_J(cd) \right)_{0,0} \tag{A.9}$$

where we use the notatation $[x] = \sqrt{2x+1}$, 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,

$$\hat{F}_{K,M} = \sum_{ab} F_{ab} \frac{1}{[K]} \left(\hat{a}^\dagger \times \tilde{b} \right)_{K,M} \tag{A.10}$$

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

A.3 Symmetries of matrix elements

Two-body matrix elements satisfy the following symmetries:

$$\begin{aligned} V_J(ab,cd) &= -(-1)^{j_a+j_b+J} V_J(ba,cd) \\ &= -(-1)^{j_c+j_d+J} V_J(ab,dc) \;\; = (-1)^{j_a+j_b+j_c+j_d} V_J(ba,dc). \end{aligned} \tag{A.11}$$

Including isospin,

$$\begin{aligned} V_{JT}(ab,cd) &= -(-1)^{j_a+j_b+J+1+T} V_{JT}(ba,cd) \\ &= -(-1)^{j_c+j_d+J+1+T} V_{JT}(ab,dc) = (-1)^{j_a+j_b+j_c+j_d} V_{JT}(ba,dc). \end{aligned} \tag{A.12}$$

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]:

$$\left(\hat{F}_{KM} \right)^\dagger = (-1)^M \hat{F}_{K,-M}. \tag{A.13}$$

For non-charge-changing transitions, Eq. (A.13) implies $F_{ab} = (-1)^{j_a - j_b} F^*_{ba}$.

Appendix B

A summary of options

BIGSTICK is a menu-driven code, to make it easier for novices. Here we write out the options (as of **version 7.11.4**), and point to where to find more information.

B.1 Main menu

When you initiate BIGSTICK, the initial menu provides you with the most used options.

```
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  *                                                                           *
  *               OPTIONS (choose one)                                        *
  * (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 (isospin fmt)    *
  * (2) Two-body density from previous wfn (default p-n format)               *
  * (x) eXpectation value of a scalar Hamiltonian (from previous wfn)         *
  * (o) Apply a one-body (transition) operator to previous wfn and write out  *
  * (s) Strength function (using starting pivot )                            *
  * (g) Apply the resolvent 1/(E-H) to a previous wfn and write out           *
  * (m) print information for Modeling parallel distribution                  *
  * (l) print license and copyright information                               *
  * (?) Print out all options                                                 *
  *                                                                           *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

For more information on:
'(i)' see Section 4.4.1;
'(n)' see Section 4.4.2 and throughout this manual;
'(d)' see Sections 4.4.3,5.1 ;

‘(2)’ see Section 5.2 ;
‘(x)’ see Section 4.7.1 ;
‘(o)’ see Section 4.7.5, 5.3.2 ;
‘(s)’ see Section 5.3 ;
‘(g)’ see Section 5.4;
‘(m)’ see Section 8.1.5;
Option ‘(l)’ simply prints out the license information from Section 1.2.

To give the full and exhaustive menu, enter ‘(?)’ to get:

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                                             *
 *               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       *
 * (ne) Compute energies but NO observable (i.e. J or T)                       *
 * (np) Compute spectrum starting from prior pivot                             *
 * (d) Densities: Compute spectrum + all one-body densities (isospin fmt)      *
 * (dx[m]) Densities: Compute one-body densities from previous run (.wfn)      *
 *     optional m enables mathematica output                                   *
 * (dp) Densities in proton-neutron format                                     *
 * (dxp) Compute one-body densities from prior run (.wfn) in p-n format.       *
 * (db) Write one-body densities to a binary file                              *
 * (dxb) Compute one-body densities from prior run, write to a binary file     *
 * (2) Two-body density from previous wfn (default p-n format)                 *
 * (2d) Two-body density from previous wfn, only initial=final, Jt=0           *
 * (2i) Two-body density from previous wfn ( isopin format)       *
 * (3) Normal spectrum but using three-body forces (beta version)              *
 * (p) Compute spectrum + single-particle occupations,(ps) to supress wfn      *
 * (occ) single-particle occupations (from previous wfn)                       *
 * (x) eXpectation value of a scalar Hamiltonian (from previous wfn)           *
 * (o) Apply a one-body (transition) operator to previous wfn and write out    *
 * (s),(sn) Strength function (using starting pivot ) wfn out normalized       *
 * (ss) Strength function (using starting pivot ), but no output wfn or J,T    *
 * (su) Strength function (using starting pivot ) wfn out unnormalized         *
 * (sb) Strength function (using block of starting pivots )                    *
 * (sbs) Strength function (using block of starting pivots ) no wfn out        *
 * (a) Apply a scalar Hamiltonian to a previous wfn and write out              *
 * (h) Compute matrix elements of a scalar Hamiltonian (inputs as basis)       *
 * (g) Apply resolvent 1/(E-H) to a previous wfn and write out                 *
 * (gv) Apply resolvent 1/(E-H) to a previous wfn, then take dot prod          *
 * (gc) Apply resolvent 1/(E-H) to a previous wfn, E complex, and write out    *
 * (v) Overlap of initial states with final states                             *
 * (pv) Read in previous vector, write out Lanczos coef, take dot prod         *
```

```
 * (m) print information for Modeling parallel distribution               *
 * (md) Modeling parallel distribution for 1-body densities               *
 * (m0) Compute dimensions only                                           *
 * (t) create TRDENS-readable file for post processing                    *
 * (tx) create TRDENS-readable file for post processing from previous wfn *
 * (tw) from TRDENS-readable file create standard BIGSTICK .wfn file      *
 * (b) Create binary file with full basis information (for postprocessing) *
 * (ba) Create ASCII file with full basis information (for postprocessing) *
 * (wh) write out Hamiltonian matrix to a file and stop                   *
 * (wo) write out one-body transition matrix to a file and stop           *
 * (co) Compute configurations (partitions)                               *
 * (cx) Compute configurations (partitions) from prior wfn                *
 * (jp) Project states of good J from prior wfns and normalize            *
 * (ro) Read in multiple files of wfns and orthonormalize                 *
 * (ru) Read in multiple files of wfns but DO NOT orthonormalize          *
 * (c) Compute traces                                                     *
 * (l) print license and copyright information                            *
 * (?) Print out all options                                              *
 *                                                                        *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

‘(ne)’, ‘(ns)’, see section 4.4.2;
‘(np)’, see Sections 4.6.2, 4.7, 7.3.1;
‘(dx[m])’, ‘(dxp)’ see Section 4.4.3;
‘(db)’, ‘(dxb)’ see Section 4.4.3;
‘(p)’, ‘(ps)’ see Section 4.4.4;
‘(occ)’, see Section 4.4.4;;
‘(sn)’, ‘(ss)’, see Section 4.7.8;
‘(sb)’, ‘(sbs)’ see Section 4.7.8, 7.3.2;
‘(a)’, see Section 4.7.6;
‘(h)’, see Section 4.7.2
‘(gv)’,‘(gc)’, see Section 5.4;
‘(v)’, see Section 4.7.9;
‘(pv)’, see Section 5.4;
‘(t)’, ‘(tw)’, ‘(tx)’, see Section 6.2;
‘(b)’, ‘(ba)’, see Section 6.2.1;
‘(wh)’, ‘(wo)’, see Section 6.3;
‘(cv)’,‘(cx)’ see Section 4.5.3;
‘(jp)’, see Section 4.7.3;
‘(ro)’, ‘(ru)’, see Section 4.7.4 ;
‘(c)’, see Section 4.5.2.

B.2 Diagonalization menu

If one is finding eigenpairs, then after the initial setup, which include constructing the basis and reading in the Hamiltonian, one chooses a method of finding the eigenpairs. See also Sec. .

```
/ ------------------------------------------------------------------------\
|                                                                         |
|    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                              |
|                                                                         |
| (bd) Block Lanczos with default convergence (STANDARD)                  |
| (bf) Block Lanczos with fixed (user-chosen) iterations                  |
| (bc) Block 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       |
| (tb) Thick-restart block Lanczos with default convergence               |
|                                                                         |
| (sk) Skip Lanczos (only used for timing set up)                         |
| (li) Lanczos iterations only, no further eigensolutions                 |
|                                                                         |
\ ------------------------------------------------------------------------/
```

'`(ex)`' This option will use Householder to find all eigenpairs, although you can choose to write out only the lowest N. Recommended for basis dimensions of small numbers (< 100), can work easily for up to dimensions of around 1000, and can be applied, with increasing time, up to dimensions $< 10,000$. Note that Householder scales like (dimension)^3.

The most common options to use are standard Lanczos (or vector Lanczos, to distinguish from block Lanczos).

'`(ld)`' is the most common choice. For a discussion of the default convergence criteria, see Sec. 4.6.1.

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

One can also set a fixed number of iterations; this can be useful for timing purposes and for cases of tricky convergence.
'`(lf)`'

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

It is possible to choose your own convergence criteria. See also Sec. 4.6.1.
‘(lc)’

```
      Enter nkeep, max # iterations for lanczos

  Enter how many ADDITIONAL states for convergence test
  ( Defaul t=            5 ; you may choose 0 )

     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;

  Enter desired tolerance
 (default tol =  0.100E-02 )
```

The next set of options are block Lanczos. Instead of the Hamiltonian matrix acting on a single Lanczos vectors, it acts on a block of Lanczos vectors. This leads to efficiencies, as the cost of constructing a Hamiltonian matrix element is amortized across the application to more than one vector. On the other hand, if one starts with a random block, one needs more iterations than in standard Lanczos. The solution is to use bootstrapped block Lanczos (Zbikowski and Johnson [2023]), reading in an approximate solution. To read in a block of vectors, use option ‘(p)’ in the initial menu. For more details see Sec. 4.6.2.

Yet another but useful alternative is thick-restart Lanczos(Wu and Simon [2000]), described in Sec. 7.2. Thick-restart requires more iterations, but the amount of storage for Lanczos vectors, as well as reorthogonalization time, is greatly reduced. This is particularly useful for very large dimension cases on limited systems.

The final options are rather specialized:
‘(sk)’ This option is only used if you want to know how much time is being used in set-up.
‘(li)’ Again, for timing purposes, will carry out Lanczos iterations but not find the eigenpairs. The Lanczos coefficients will be written out to a file with extension .lcoef.

```
  Lanczos iterations ONLY to get Lanczos coefficients
  Enter # iterations for lanczos
```

Appendix C

Troubleshooting

BIGSTICK is a large and complex code, designed to run flexibly on platforms from laptops up to leading-edge supercomputers. While we have tried to make it as robust and user-friendly as practical, given that it is a code primarily for cutting-edge research, and only secondarily for pedagogy, it is easy to make mistakes or get confused.

C.1 Overall

- Read this manual! It contains much valuable information and many valuable hints.
- Try the sample runs provided.
- We strongly encourage you to try some small, simple cases and gradually build your way up. No one goes directly from integrating $f(x) = x$ to performing contour integrals in the complex plane. By taking the time to run cases of increasing complexity you will build up your familiarity with the capabilities of BIGSTICK.
- Read the output *carefully*. If it fails to recognize an input, or an input file, it will try to tell you. Also read the log file (either `XXX.log` or `bigstick.logfile` depending whether or not you gave an output name `XXX`). The log file also contains much valuable information and not infrequently warnings of problems or potential problems.

C.2 Compilation

- Note that the default compiler is current the Intel `ifort` compiler. This is the compiler for `make serial`, `make openmp`, etc. There are compiler options for `gfortran` which is widely available, including `make gfortran-openmp`.

To the best of our working knowledge, however, `gfortran` does not have a straightforward implementation with MPI.

- If you switch compiler options, i.e., go from `make serial` to `make openmp`, or from `make gfortran` to `make gfortran-openmp`, you must recompile from scratch. Do a `make clean` to remove all intermediate object and module files.

C.3 Inputs

- **Interaction files**. It is fairly easy to make a mistake entering the interaction file information. Remember that, broadly speaking, we have two main types of interaction files, which we chose to follow the format of other, widely used codes: `OXBASH/NuShell`-type files, and `MFDn`-style files. In general, the `OXBASH/NuShell`-type files **must have a name with the extension `.int`, but you only enter the name, not the extension**. Conversely, for `MFDn`-style files, **you must enter the full name, even if it has a .int extension**.

  For example, if you have an interaction file `usdb.int`, which is in `OXBASH/NuShell`-type format, you must enter

  ```
  usdb
  ```

  when asked for the file name. If you enter `usdb.int`, the code will attempt to interpret it as an `MFDn`-format. It is acceptable to have a `MFDn`-file format with an extension, for example, `MyLittleInteraction.int`, but you **must enter the full name for the file to be read correctly**.

- Note that `OXBASH-NuShell`-format files include isospin-format, and both normalized and unnormalized proton-neutron formats. See section 4.3 for more details. `OXBASH-NuShell`-format includes single-particle energies, while `MFDn`-format do not.

- For best results, make sure all input files refer to exactly the same single-particle space. In particular, the interaction file single particle spaces should match those from the `.sps` file or the `auto` option when defining the single-particle space. If you are not sure, read the output from `BIGSTICK` as it attempts to read the file; it will tell you what it thinks the model space is.

C.4 Large cases and parallel computing

- We encourage you to use the modeling option (option '`m`') from the main menu before beginning a large parallel run, to determine the memory requirements. You can run this in MPI mode itself, on a small number of

processors, which for large no-core shell model calculations can speed up the modeling.

- if you fail to get a distribution, either in modeling or in an actual run, try increasing the number of processors. You may also need to increase the memory available, for example by using more OpenMP threads; this makes more memory available to each MPI process. Also, you may try deleting the file `timinginfo.bigstick`; this file keeps track of the timing of the code on a particular machine and problem, but can in some cases become corrupted.

- The real computational burden for any configuration interaction code is not the basis dimension, but the number of nonzero matrix elements. This is roughly the number of operations as computed by `BIGSTICK`, though not exactly, as diagonal matrix elements can require multiple operations. The factorization algorithm used by `BIGSTICK` and similar codes reduces the memory burden relative to codes that store the nonzero matrix elements, such as `MFDn`, but the time for a matrix-vector multiplication is still the same.

C.4.1 Proton-neutron imbalance

`BIGSTICK`'s efficiency stems from factorizing the proton-neutron partitioning. The downside is that if one has a large problem with a significant proton-neutron imbalance, the efficiency of `BIGSTICK` is impaired and, for the largest cases, may not even run. So, for example, a no-core calculation of ^{29}F, with $Z = 9$ and $N = 20$, at high $N_{\max}$ becomes bogged down. We are continuing to research approaches to mitigate such problems, but the solution is not simple. Unfortunately, at this time, simply adding MPI ranks does not provide an easy solution.

If, for example, you are carrying out a single-species calculation, such as computing tin isotopes with a ^{100}Sn core, `BIGSTICK` cannot factorize the problem. Instead, effectively all matrix elements are stored in memory. (Technically, the problem is even worse, because diagonal matrix elements have many contributing operations; see Section 4.2.1.) If you are doing a full-configuration calculation, however, one can mitigate the problem by artificially breaking up the model space using the weighting truncation (cf. section 4.2.2). Assign one of the single-particle orbits a different value of W from the rest; allow for truncations, but then choose the maximum allowed truncation. This is not a perfect solution, but it does seem to prevent certain error messages, notably about 'non-contiguous introns' (a strategy to minimize the storage of jumps). One should not overdo an artificial break-up of the space. Using W to break up the space slows down the code, and too fine-grained a break-up will make it run unacceptably slow, or not at all. Experience suggests a reasonable strategy is to select the orbital with the largest j to have a different W.

`BIGSTICK` is both flexible and complex, so the best solution is not always obvious. In particular, valence-space calculations, where one has many particles in

a limited single-particle space, are very different from no-core calculation, where one typically has a few particles in a much larger single-particle space. You are encouraged to experiment with different configurations to find an appropriate one.

C.4.2 Important clues in a failure

Sometimes a run will fail. While experience is the best guide to what has gone wrong in a calculation, here are some things you can look at. An experienced computational physicist will know that the final error message is seldom the entire story, and that one typically needs to look upstream to discover the problem.

The two main failures are, first, the program crashes or otherwise does not run, and second, the results are strange in some way.

- **Basis dimension**. Though not the most important quantity, certainly the basis dimension can give signals as to potential problems. The basis dimension is always printed out:

  ```
  Total basis =              177070720
  ```

  If the dimension is small, say fewer than a few hundred, it is better to avoid Lanczos and use "exact" (i.e., full) diagonalization. If the basis dimension is less than a few million, you do not need to use MPI. (In any case, using many MPI ranks for small cases is counterproductive.) If the basis dimension is greater than a few tens of millions, MPI is often more efficient, and if greater than 100 million, likely necessary. Note that you can quickly compute the basis dimension *only* through the option '`(m0)`.' Furthermore, dimension of the order of ten billion starts to reach the limits of the code on modern supercomputers.

  There is no rigid rule as to how many MPI ranks to assign. It depends upon many factor: how close is N to Z ($N = Z$ yields the most efficient factorization in terms of matrix element storage), the choice of many-body truncation (large '`max excite`' tend to be less efficient), and so on. Our recommendation is always to try some smaller cases first and work your way up to your target.

- **Jump storage**. More important than basis dimension is the storage of the data for non-zero Hamiltonian matrix elements. These are stored as 'jumps' and again this information is output:

  ```
   RAM for jumps in storage    (total)  :   675.022 Mb
  ```

  In MPI mode, the code will attempt to distribute these jumps.

  ```
  RAM for jumps in storage    (total)     :   32586.691 Mb
   Max RAM for local storage of jumps     :    4693.353 Mb
  ```

BIGSTICK sets a ceiling for how much memory per MPI rank will be assigned to jumps. This is the constant maxjumpmemory_default which is set in the file bmodules_flags.f90

```
real    :: maxjumpmemory_default  = 64.0
```

You can change this (and recompile), but be sure you system has this much memory per MPI rank! This is where hybrid MPI+OpenMP is particularly useful: you can assign multiple OpenMP cores (threads) to each MPI rank in order to build up sufficient memory. Other solutions include changing the number of MPI ranks as well as changing the size of the fragments. In some cases, a slight change will allow the code to find a better distribution of the work.

- **Bad values of J.** BIGSTICK is an M-scheme code, which means that total M or J_z of the basis is fixed. Because it is assumed that the Hamiltonian is rotationally invariant–and, indeed, significant changes to the code would be necessary to violate this assumption–one can have simultaneous eigenstates of the Hamiltonian and of $\hat{J}^2$. In general, therefore, converged states should have "good" values of J, that is, integer for even numbers of particles and half-integer for odd-numbers. The only exception is when one has degeneracies, states of different J with the same energy. (BIGSTICK does not automatically separate there.) Unless one is working with, for example, an algebraic or schematic interaction, such as pure pairing/seniority or $Q \cdot Q$, this should not happen.

- **Missing/wrong format Hamiltonian file**. If the input Hamiltonian file is formatted incorrectly, clashes with the single-particle space, or simply not read in, you can get strange answers, such as non-integer/non-half-integer values of J. This can particularly happen if there is a mismatch between the defined single-particle space and the Hamiltonian file, if one is using either the default '`iso`' format (which is inherited from previous codes) or the similar proton-neutron formats '`xpn`' or '`upn`.' (The '`mfd`' format does not have this problem, but it is still possible to have such files with matrix elements missing.) The code will try to check, and you should pay attention. You will see such information as

  ```
  * * NOTICE: I expect single-particles space with  3  orbits
  ```

  and

  ```
  As a check, first two-body matrix element in list is   -1.3796
  ```

- **Too few/too many Lanczos iterations**. If you have too few Lanczos iterations, the solutions may not converge. One should avoid requesting too many Lanczos iterations. For one, this can overwhelm the storage of Lanczos vectors. When you initiate Lanczos, you should see

```
Internal storage of all lanczos vectors =    2.79135983E-02  Gb
```

this is the storage per MPI rank. You will have to gauge how much storage per MPI rank you have available.

Another, rare situation can occur in cases with highly degenerate spectra. In that case, one can have the Lanczos parameter $\beta_i \approx 0$. This can lead to a divide-by-zero problem. (See section 7.1 for more on the Lanczos algorithm.) `BIGSTICK` tries to restart, but if β_i is merely very small, and the ratio β_i/β_{i-1} is not in a designated range, it may fail to do. In that case the new Lanczos vector is incorrect and one can get very strange values. There is no simple solution for this; either carefully monitor the Lanczos procedure, or add a small random Hamiltonian to slightly split degenerate solutions.

C.5 If you want to contact us with a problem

We welcome feedback on `BIGSTICK`, including bugs. If you are having difficulty, it is best to send us as much information as possible: send us a complete copy of the output (not just an error message out of context–`BIGSTICK` will often print out information which may be helpful), the log file (either `XXX.log` or `bigstick.logfile` depending whether or not you gave an output name `XXX`), as well as your input files and the input commands or scripts you used. Most of the time the mistakes made are simple ones, arising from simply not understanding the inputs; as noted, this is a complex code, so it is easy to make a mistake.

Appendix D

Glossary

BIGSTICK is a big code with complex algorithms, and in the code itself and while running, one can find some unusual terms of art. While you do not need to know all these words, we explain here some of the specialized terms

- **Jumps**. These are the basic data to enable reduced storage of the Hamiltonian. Initially devised by Caurier and collaborators, jump arrays store the action of particle-number conserving n-body operators, for $n = 1$ ($\hat{a}^\dagger\hat{a}$), or $n = 2$ ($\hat{a}^\dagger\hat{a}^\dagger\hat{a}\hat{a}$). ($n = 3$ exists, but is currently under refurbishing.) Jumps are used to reconstruct the Hamiltonian. For proton-proton or neutron-neutron interactions, jumps are just the matrix elements; but for proton-neutron interactions, one combines a proton one-body jump with a neutron one-body jump to get a two-body proton-neutron two-body interaction. This is the source of BIGSTICK's efficiency.
- **Fragments**. (Only used in MPI, that is, distributed memory.) For large dimension spaces, one breaks up the active vectors for matrix-vector (or matrix-matrix) multiplication into fragments. Contrary to common assumptions, one should try to keep the fragments as large as possible; this makes the reconstruction of the Hamiltonian matrix elements from jumps more efficient.
- **Sterile orbitals**. A weight of 99 in the `.sps` file signals a 'sterile' orbital which is not used. This is a away to help define different proton and neutron valence spaces.
- **Species**. BIGSTICK assumes fermions come in two distinguishable species, usually protons and neutrons, although it can also be spin-up and spin-down fermions.

Here are some additional terms which, while you are unlikely to ever need to know, help explain some of how BIGSTICK works.

- **Opbundles**. Opbundles are collections of jumps between the same initial and final sectors.

- **Hops**. These are similar to jumps, but are more primitive particle addition $\hat{a}^\dagger$ or removal $\hat{a}$. Used to construct jumps. Hops are between *haikus*.

- **Haikus**. The basis is constructed from occupation-number representations of Slater determinants. However the only actual storage is in 'half-Slater determinants,' or *haikus*. One has 'left' haikus, constructed from single-particle states with $m < 0$, and 'right' haikus, constructed from single-particle states with $m \geq 0$. (In the nuclear case, values of m are half-integers, divided into protons and neutrons. In atomic cases, however, one has spin-up and spin-down particles, and then m refers to the orbital angular momentum and is an integer. This latter is not used very much.)

- **Pieces**. (Only used in MPI, that is, distributed memory.) When not active, that is, part of the Hamiltonian multiplication process, the Lanczos vectors are stored across multiple MPI ranks. These vectors are broken up as 'pieces.' The same piece of each Lanczos vector is stored on the same MPI rank. Pieces are much smaller than fragments and function differently.

- **Sectors**. A sector is a grouping of Slater determinants of a given species that have the same quantum numbers, namely M, parity, and W.

- **Blocks**. A block is a grouping of haikus (of a given species) that that the same quantum numbers, namely M, parity, W, and particle number.

- **Conjugate**. `BIGSTICK` achieves efficiency through quantum numbers which control how to combine data. For example, in generating the basis, the quantum numbers of the proton Slater determinants and the quantum numbers of the neutron Slater determinants must combine to a fixed result, such as total M, parity, or up to some maximum total W. By grouping Slater determinants into sectors, and at a lower level, haikus into blocks, one can set up relatively simple and efficient loops. For a sector or block with some set of quantum numbers, the conjugate sectors or blocks are those that are allowed to combine with it.

Appendix E

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 [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.

Several papers and conference proceedings describe our work on `BIGSTICK`: [Johnson et al., 2013, Shan et al., 2015, 2017, 2018], as well as this manual, whose original citation is [Johnson et al., 2018].

Some groups besides ours use `BIGSTICK` in their research; for some recent examples see Kruppa et al. [2021], Rodkin and Tchuvil'sky [2021], Romero et al. [2021], Cirigliano et al. [2022], Romero et al. [2022], Haxton et al. [2024], Brökemeier et al. [2025].

Bibliography

F. Andreozzi and A. Porrino. *J. Phys. G: Nucl. Part. Phys*, 27:845, 2001.

B. R. Barrett, P. Navrátil, and J. P. Vary. Ab initio no core shell model. *Progress in Particle and Nuclear Physics*, 69:131–181, 2013.

F. Brökemeier, S. M. Hengstenberg, J. W. T. Keeble, C. E. P. Robin, F. Rocco, and M. J. Savage. Quantum magic and multipartite entanglement in the structure of nuclei. *Phys. Rev. C*, 111:034317, Mar 2025.

B. Brown, A. Etchegoyen, and W. Rae. Computer code OXBASH: the Oxford University-Buenos Aires-MSU shell model code. *Michigan State University Cyclotron Laboratory Report No. 524*, 1985.

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 W. A. Richter. New "usd" hamiltonians for the *sd* shell. *Phys. Rev. C*, 74:034315, Sep 2006.

B. A. Brown and B. H. Wildenthal. Status of the nuclear shell model. *Annual Review of Nuclear and Particle Science*, 38:29–66, 1988.

P. Brussard and P. Glaudemans. *Shell-model applications in nuclear spectroscopy*. North-Holland Publishing Company, Amsterdam, 1977.

E. Caurier and F. Nowacki. Present status of shell model techniques. *Acta Physica Polonica B*, 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.

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.

V. Cirigliano, K. Fuyuto, M. J. Ramsey-Musolf, and E. Rule. Next-to-leading order scalar contributions to $\mu \to e$ conversion. *Phys. Rev. C*, 105:055504, May 2022.

D. B. Cook. *Handbook of computational quantum chemistry*. Oxford University Press, 1998.

A. De-Shalit and I. Talmi. *Nuclear shell theory*, volume 14. Academic Press, 2013.

J. Draayer, T. Dytrych, K. Launey, and D. Langr. Symmetry-adapted no-core shell model applications for light nuclei with qcd-inspired interactions. *Progress in Particle and Nuclear Physics*, 67(2):516–520, 2012.

A. R. Edmonds. *Angular momentum in quantum mechanics*. Princeton University Press, 1996.

D. Gloeckner and R. Lawson. Spurious center-of-mass motion. *Physics Letters B*, 53(4):313–318, 1974.

O. C. Gorton. *Shell Model Methods, Statistical Nuclear Reactions, and Beta-delayed Neutron Emission*. PhD thesis, University of California, Irvine, 2024.

W. Haxton, K. McElvain, T. Menzo, E. Rule, and J. Zupan. Effective theory tower for $\mu\rightarrow$ e conversion. *Journal of High Energy Physics*, 2024(11):1–59, 2024.

K. L. Heyde. *The nuclear shell model*. Springer, 1994.

F. Jensen. *Introduction to computational chemistry*. John Wiley & Sons, 2017.

C. W. Johnson. Spin-orbit decomposition of *ab initio* nuclear wave functions. *Phys. Rev. C*, 91:034313, Mar 2015.

C. W. Johnson, W. E. Ormand, and P. G. Krastev. Factorization in large-scale many-body calculations. *Computer Physics Communications*, 184:2761–2774, 2013.

C. W. Johnson, W. E. Ormand, K. S. McElvain, and H. Shan. Bigstick: A flexible configuration-interaction shell-model code. *arXiv preprint arXiv:1801.08432*, 2018.

P. Knowles and N. Handy. A new determinant-based full configuration interaction method. *Chemical physics letters*, 111(4-5):315–321, 1984.

A. Kruppa, J. Kovács, P. Salamon, and Ö. Legeza. Entanglement and correlation in two-nucleon systems. *Journal of Physics G: Nuclear and Particle Physics*, 48(2):025107, 2021.

R. Lawson. *Theory of the nuclear shell model*. Clarendon Press Oxford, 1980.

P.-O. Löwdin. Quantum theory of many-particle systems. I. Physical interpretations by means of density matrices, natural spin-orbitals, and convergence problems in the method of configurational interaction. *Phys. Rev.*, 97:1474–1489, Mar 1955.

P. Navrátil, J. Vary, and B. Barrett. Large-basis ab initio no-core shell model and its application to 12 c. *Physical Review C*, 62(5):054311, 2000.

F. Palumbo. Intrinsic motion and translational invariance in shell-model calculations. *Nuclear Physics A*, 99(1):100–112, 1967.

F. Palumbo and D. Prosperi. Effects of translational invariance violation in particle-hole calculations. application to 208pb. *Nuclear Physics A*, 115(2): 296–308, 1968.

T. Papenbrock and D. J. Dean. Factorization of shell-model ground states. *Phys. Rev. C*, 67:051303, May 2003.

T. Papenbrock and D. J. Dean. Density matrix renormalization group and wavefunction factorization for nuclei. *Journal of Physics G: Nuclear and Particle Physics*, 31(8):S1377, 2005.

T. Papenbrock, A. Juodagalvis, and D. J. Dean. Solution of large scale nuclear structure problems by wave function factorization. *Phys. Rev. C*, 69:024312, Feb 2004.

B. N. Parlett. *The symmetric eigenvalue problem*, volume 7. SIAM, 1980.

W. H. Press, B. Flannery, S. Teukolsky, and W. Vetterling. *Numerical recipes in fortran*. Cambridge university press, 1992.

D. M. Rodkin and Y. M. Tchuvil'sky. Detailed theoretical study of the decay properties of states in the ^{7}He nucleus within an ab initio approach. *Phys. Rev. C*, 104:044323, Oct 2021.

A. Romero, J. Engel, H. L. Tang, and S. E. Economou. Solving nuclear structure problems with the adaptive variational quantum algorithm. *arXiv preprint arXiv:2203.01619*, 2022.

A. M. Romero, J. M. Yao, B. Bally, T. R. Rodríguez, and J. Engel. Application of an efficient generator-coordinate subspace-selection algorithm to neutrinoless double-β decay. *Phys. Rev. C*, 104:054317, Nov 2021.

H. Shan, S. Williams, C. Johnson, K. McElvain, and W. E. Ormand. Parallel implementation and performance optimization of the configuration-interaction method. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2015.

H. Shan, S. Williams, C. Johnson, and K. McElvain. A locality-based threading algorithm for the configuration-interaction method. 2017. URL `http://escholarship.org/uc/item/9sf515zf`.

H. Shan, S. Williams, and C. W. Johnson. Improving mpi reduction performance for manycore architectures with openmp and data compression. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 1–11. IEEE, 2018.

I. Shavitt. The history and evolution of configuration interaction. *Molecular Physics*, 94:3–17, 1998.

C. D. Sherrill and H. F. Schaefer. The configuration interaction method: Advances in highly correlated approaches. *Advances in quantum chemistry*, 34: 143–269, 1999.

N. Shimizu. Nuclear shell-model code for massive parallel computation," kshell". *arXiv preprint arXiv:1310.5431*, 2013.

N. Shimizu, T. Mizusaki, Y. Utsuno, and Y. Tsunoda. Thick-restart block Lanczos method for large-scale shell-model calculations. *Computer Physics Communications*, 244:372–384, 2019.

P. Sternberg, E. Ng, C. Yang, P. Maris, J. Vary, M. Sosonkina, and H. V. Le. Accelerating configuration interaction calculations for nuclear structure. *The Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

J. Suhonen. *From Nucleons to Nucleus: Concepts of Microscopic Nuclear Theory*. Springer Science & Business Media, 2007.

I. Talmi. *Simple models of complex nuclei*. CRC Press, 1993.

J. Toivanen. Efficient matrix-vector products for large-scale nuclear shell-model calculations. *arXiv preprint arXiv:nucl-th/0610028*, 2006.

I. S. Towner. *A shell model description of light nuclei*. Clarendon Press Oxford, 1977.

D. A. Varshalovich, A. N. Moskalev, and V. K. Khersonskii. *Quantum theory of angular momentum*. World scientific, 1988.

A. W. Weiss. Configuration interaction in simple atomic systems. *Phys. Rev.*, 122:1826–1836, Jun 1961.

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.

K. Wu and H. Simon. Thick-restart Lanczos method for large symmetric eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(2): 602–616, 2000.

R. M. Zbikowski and C. W. Johnson. Bootstrapped block Lanczos for large-dimension eigenvalue problems. *Computer Physics Communications*, 291: 108835, 2023.