MY DISSERTATION TITLE

UNDERGRADUATE RESEARCH THESIS

Presented in Partial Fulfillment of the Requirements for Graduation
with Honors Research Distinction in Physics in the College of Arts
and Sciences of The Ohio State University

By

Matthias Heinz

Undergraduate Program in Physics

The Ohio State University

2018

Thesis Committee:

Professor Richard Furnstahl, Advisor

Prof. Richard Furnstahl

Prof. Robert Perry

Prof. P Sadayappan

Abstract

An abstract goes here. It should be less than **500 words**.

Table of Contents

List of Figures

List of Tables

Chapter 1
INTRODUCTION

Nuclear theory is concerned with modeling atomic nucleus and nuclear matter. Nuclear matter is made of protons and neutrons, which are generally referred to as nucleons. Nucleons interact primarily through the strong interaction, which together with the weak interaction governs the stability of nuclear isotopes. Nuclear theory seeks to answer open questions in four areas: what are the properties of nuclei, especially exotic isotopes, what are the properties of nuclear matter in astrophysical systems, what can be learned about beyond standard model (BSM) physics from the nucleus, and how can more accurate models of nuclear systems be leveraged in a variety of applications, ranging from national security and energy to medicine.

Nuclear theory faces two major challenges when trying to model systems of nucleons at low energies. The first is that this is a quantum many-body problem. Quantum many-body theory is relevant to many different fields, including quantum chemistry and condensed matter theory. Many-body problems quickly become intractable when approached naively due to the combinatorial growth of the size of the problem with respect to the number of particles. The second major challenge of nuclear theory is that the while the forces between nuclei are given by quantum chromodynamics (QCD), the theory underlying the strong interaction, QCD cannot be solved in closed form at energies relevant to nuclei. The strong interaction is between quarks and glu-

ons, collectively referred to as partons, which make up nucleons, instead of between nucleons themselves. These challenges forced nuclear physics in the past to rely on phenomenological models, both for the form of the strong interaction and the treatment of the many-body problem. These models were typically fit to experimental data for certain nuclei and used to predict the properties of nearby nuclei. However, their predictive ability did not extend far outside the domain in which they were fit, limiting their range of applicability. Furthermore, they did not provide theorists with error estimates or any ways to tell when they became invalid.

With the Facility for Rare Isotope Beams (FRIB) on the horizon, there is demand for more accurate calculations of nuclear observables with new methods as well as for theoretical predictions of properties of heavy, exotic nuclei that have not been measured yet. Moreover, accurate models of nuclear matter are essential to understanding dense stars, supernovae, and certain astrophysical events, such as the collisions of neutron stars, which are hypothesized to be a location for the synthesis of heavy elements, such as gold and silver. Improved nuclear models are also essential in predicting the rate of certain nuclear decays, such as neutrino-less double-beta decay, which is currently being searched for by many experiments to answer whether neutrinos are Majorana particles, which means they are their own anti-particle, or not.

In recent years, the improvements in computational hardware and the development of new computational methods has brought about a rapid expansion in the range of nuclear isotopes able to be modeled via ab-initio calculations, or calculations starting from the forces between nucleons as determined by scattering experiments and few-body nuclei. The improvements in computational hardware have come through the development and proliferation of high-throughput devices (such as GPUs) and the assembly of highly parallel systems. Current trends suggest that a machine with

exa-FLOP (floating-operations per second) throughput will exist by 2020 and such systems will be commonplace soon after. Much work is being done to ensure that the processing, networking, and power-consumption of these systems will continue to scale as it has for the past two decades.

The computational methods used in modern low-energy nuclear theory come in three classes: interaction models, renormalization group (RG) methods, and effective field theory (EFT) methods. Modern interaction models, like no-core shell model (NCSM), seek to model nuclei from first principles and work in conjunction with RG and EFT methods to make these calculations feasible. EFT methods offer systematically improvable, model-independent ordering of components of some interaction that allows for importance truncation and uncertainty quantification. RG methods are applied to interactions given by EFT methods and modify these interactions to match the resolution relevant to the problem at hand. These methods have already been used to offer both theoretical predictions that improve on previous calculations and theoretical predictions for nuclei that were unable to be modeled previously. The use of these methods in low-energy nuclear calculations is the state of the art. As a result, questions regarding their application are open problems and the focus of a lot of research.

The similarity renormalization group method (SRG) is an interaction softening method from the class of RG methods whose use in nuclear theory was first explored in a 1-dimensional setting at OSU in 2007. It is a class of continuous unitary transformations that decouple large off-diagonal matrix elements in the interaction Hamiltonian, softening the potential as a result. The evolution of the Hamiltonian to a decoupled form allows a truncated subspace of the original basis to be used in later calculations without affecting the observables which are the lowest energy eigenvalues. This basis truncation offers a significant reduction in the size of the problem. SRG

is frequently used in modern nuclear theory calculations to soften interactions and extend the range of certain calculations to larger systems. In our work, we return to a simple 1-dimensional setting to study features of SRG and seek to understand how to optimize its use in many-body calculations.

1.1 Contributions

Our main contributions are:

- We have developed an open-source Python library with easy to use abstractions for the SRG method for use by others to test SRG in a simple setting.

- We verify the results published in the 2009 Jurgenson paper for 2-body and 3-body systems, suggesting the correctness of our implementation. We also provide a suite of tests for the library, verifying that it fails gracefully when misused and correctly calculates the results simple analytically solvable cases when used correctly.

- We explore some alternative transformation generators and seek to quantify their performance relative to the standard generator that is currently used. We discuss the results of these calculations and offer some preliminary analysis.

1.2 Outline

The rest of the thesis is as follows:

- In chapter 2, we review the matrix representation of quantum operators. We then discuss the details of two different bases used, relative Jacobi momentum coordinates and 1-dimensional symmetrized harmonic oscillator states. We then discuss the details of the SRG method and explain the need for a careful revisit of SRG in a 1-dimensional setting.

- In chapter 3, we explain the design of the Python library. Much of the effort on this project went into making the library design logical and simple-to-use, so this chapter will explain the abstractions made and the reasoning behind those decisions.

- In chapter 4, we discuss ???

- In chapter 5, we discuss ???

Chapter 2
SIMILARITY RENORMALIZATION GROUP FORMALISM

In this chapter, we introduce the physics concepts and formalism necessary to understand the 1-dimensional problems to which SRG is applied. We then introduce SRG and discuss details and open questions regarding its use.

2.1 Quantum Mechanics Operators

In quantum mechanics, there are operators, which are linear operators corresponding to some physical quantity. Examples of operators include the Hamiltonian, $\hat{H}$, which is the sum of kinetic and potential energy operators, $\hat{T}$ and $\hat{V}$ respectively, and represents all possible energies of the system. These operators act on state vectors, which in Dirac notation are denoted by abstract kets, $|\Psi\rangle$. State vectors represent a specific state of the system. A state, $|\Psi\rangle$, is said to be an eigenstate with eigenvalue ω of some operator, $\hat{\Omega}$, when the following equation holds:

$$\hat{\Omega}|\Psi\rangle = \omega|\Psi\rangle. \tag{2.1}$$

These eigenvalues are the *observables* of quantum mechanical systems.

Operators exist in a Hilbert space spanned by a basis of state vectors, $|\Psi_i\rangle$. An

operator's representation with respect to a basis is given by its matrix elements,

$$\hat{\Omega}_{ij} = \langle \Psi_i | \, \Omega \, | \Psi_j \rangle . \tag{2.2}$$

An operator can be transformed to a different basis through a unitary transformation, which preserves its eigenvalues and thus its observables. Examples of 1-dimensional single particle bases include particle position, x, particle momentum, p, and the 1-dimensional harmonic oscillator basis, which will be discussed next.

2.1.1 1-Dimensional Harmonic Oscillator

The 1-dimensional harmonic oscillator is a single particle being acted upon by a linear restoring force, $F(x) = -kx$. The Hamiltonian for the quantum harmonic oscillator is given by

$$\hat{H}_{HO} = \frac{\hat{p}^2}{2m} + \frac{1}{2} m \omega^2 \hat{x}^2. \tag{2.3}$$

which has eigenstates $|n\rangle$, which have corresponding wavefunctions

$$\Psi_n(x) = \frac{1}{\sqrt{2^n n!}} \left(\frac{m\omega}{\pi\hbar} \right)^{1/4} e^{-m\omega x^2/2\hbar} H_n \left(\sqrt{\frac{m\omega}{\hbar}} x \right) \tag{2.4}$$

$$\Phi_n(p) = \frac{(-i)^n}{\sqrt{2^n n!}} \left(\frac{1}{\pi m \hbar \omega} \right)^{1/4} e^{-p^2/2m\hbar\omega} H_n \left(\frac{p}{\sqrt{m\omega\hbar}} \right) \tag{2.5}$$

where $H_n(x)$ is the n-th Hermite polynomial. The harmonic oscillator can also be rewritten as

$$\hat{H}_{HO} = \hbar\omega \left(\hat{a}^\dagger \hat{a} + \frac{1}{2} \right), \tag{2.6}$$

where $\hat{a}^\dagger$ and $\hat{a}$ are raising and lowering operators which act on the harmonic oscillator eigenstates like

$$\hat{a}^\dagger |n\rangle = \sqrt{(n+1)} |n+1\rangle \tag{2.7}$$

$$\hat{a} \left| n \right\rangle = \sqrt{(n)} \left| n - 1 \right\rangle. \tag{2.8}$$

For an operator, $\hat{V}$, represented with respect to a momentum basis, $V(p, p')$, a transformation to a harmonic oscillator basis can be achieved by the following calculation:

$$\left\langle n \right| \hat{V} \left| n' \right\rangle = \iint V(p, p') \Phi_n(p) \Phi_{n'}(p') \, dp \, dp'. \tag{2.9}$$

2.1.2 Many-particle states

For a system of A non-interacting particles, the energy eigenstates of the system can be written as a product of the eigenstates of the single particle Hamiltonians:

$$\left| \Psi \right\rangle = \prod_{i=1}^{A} \left| \Psi_k \right\rangle_i \tag{2.10}$$

where $\left| \Psi_k \right\rangle_i$ is the k-th eigenstate of the i-th particle's single particle Hamiltonian. For N non-interacting particles in a harmonic oscillator potential, these product states are denoted by:

$$\left| n_1 n_2 ... n_A \right\rangle = \prod_{i=1}^{A} \left| n_i \right\rangle_i \tag{2.11}$$

2.2 1-Dimensional System of Bosons

The Hamiltonian for a 1-dimensional system of A identical bosons with mass m that interact via a local two-body potential is as follows:

$$\hat{H} = \frac{1}{2m} \sum_{i=1}^{A} \hat{k}_i^2 + \sum_{i=1}^{A} \sum_{j=i+1}^{A} \hat{V}(x_i - x_j). \tag{2.12}$$

In this equation, the $\hat{k}_i$ are the single-particle momenta and the x_i are the single-particle coordinates. A local potential is the function of the distance between two particles, as opposed to a function of both of their absolute coordinates.

2.2.1 Jacobi Coordinates

For local potentials, a factorization of the Hamiltonian into a center-of-mass component, which is unaffected by the potential, and a component relative to the center of mass is possible. This is achieved by a transformation to relative momentum Jacobi coordinates. These are defined to be:

$$p_i = \sqrt{\frac{i}{i+1}} \left(\frac{1}{i} \sum_{j=1}^{i} (k_j - k_{i+1}) \right) \tag{2.13}$$

with k_i being the single-particle momenta as mentioned above. It is worth noting that for a system of A particles, we only need $A - 1$ Jacobi momentum coordinates. We can get $\hat{V}$ in terms of incoming Jacobi momentum p and outgoing momentum p' by taking the following Fourier transform of $\hat{V}(x_1 - x_2)$

$$\hat{V}(p, p') = \int \hat{V}(\sqrt{2}l_1) e^{-i(p-p')l_1} \, dl_1, \tag{2.14}$$

where l_1 is the coordinate conjugate of p_1, $(x_1 - x_2)/\sqrt{2}$. Removing the center-of-mass component of the Hamiltonian, we find the new Hamiltonian to be

$$\hat{H} = \frac{1}{2\mu} \sum_{i=1}^{A-1} \hat{p}_i^2 + \sum_{i=1}^{A-1} \hat{V}(p_i, p_i'). \tag{2.15}$$

For the purposes of this work, we will be starting with a potential defined with respect to the Jacobi momentum coordinates, avoiding the process of Fourier transforming a local coordinate-space potential.

2.2.2 Transformation To Harmonic Oscillator States

In momentum space, SRG evolutions require separate treatment of 2-body, 3-body, and higher-body potentials, to avoid delta functions caused by spectator particles. Spectator particles are particles in some configuration do not interact with other particles that are interacting. To avoid the cognitive load of handling all these potentials

separately, we can make a transformation to a discrete basis. The discrete basis of choice for this project is the harmonic oscillator basis with respect to the Jacobi coordinates. This transformation can be achieved as shown in equation (2.9) (reference to equation here?). From here on out, $|n_i\rangle$ will mean the n-th harmonic oscillator state with respect to the i-th Jacobi coordinate. So for a full treatment of an N-particle system, we will need product states for $N - 1$ Jacobi coordinates,

$$|n_1 n_2 ... n_{A-1}\rangle = \prod_{i=1}^{A-1} |n_i\rangle. \tag{2.16}$$

2.2.3 Symmetrizing the Harmonic Oscillator Basis

According to the spin-statistics theorem, the state of a system of bosons must be symmetric under any permutation of the particle coordinates. We approach the problem of generating a basis of states that reflect this symmetry recursively, by first symmetrizing the 2-body system and then going from a symmetrized $A - 1$-body basis to a symmetrized N-body basis. At each stage, we must diagonalize the symmetrizer, the form of which we will show for the 2-body and 3-body cases.

For the 2-particle system, we are only working with the first Jacobi coordinate and the harmonic oscillator states corresponding to it, $|n_1\rangle$. The symmetrizer is $\hat{S} = (1 + \hat{P}_{12})/2$, where $\hat{P}_{ij}$ is the operator that exchanges the coordinates between i-th and j-th particles. Because harmonic oscillator wavefunctions are either even for even n or odd for odd n, $\hat{P}_{12} |n_1\rangle = (-1)^{n_1} |n_1\rangle$ and the symmetrizer is already diagonal with eigenvalue 0 for odd n_1 and eigenvalue 1 for even n_1. We select the states with eigenvalue 1 to create our symmetric harmonic oscillator basis for the 2-particle system.

To generate the partially symmetrized basis for the 3-body system, we take the states $|n_1 n_2\rangle$ where the possible n_1 values come from the symmetrized 2-body basis.

The general 3-body symmetrizer is governed by the permutation group S_3, generated by $\hat{P}_{12}$ and $\hat{P}_{23}$, and has the form

$$\hat{S} = \frac{1}{6}(1 + \hat{P}_{12} + \hat{P}_{23} + \hat{P}_{23}\hat{P}_{12} + \hat{P}_{12}\hat{P}_{23}\hat{P}_{12}) \qquad (2.17)$$

Since the states included in our to-be-symmetrized basis are already symmetric with respect to $\hat{P}_{12}$, the symmetrizer simplifies to $\hat{S} = (1 + 2\hat{P}_{23})/3$. The matrix elements of $\hat{P}_{23}$ in our partially symmetrized basis are

$$\langle n_1' n_2' | \hat{P}_{23} | n_1 n_2 \rangle = \delta_{N',N} \langle n_1' n_2' | n_1 n_2 \rangle_3 , \qquad (2.18)$$

where $N' = n_1' + n_2'$, $N = n_1 + n_2$, and $\langle n_1' n_2' | n_1 n_2 \rangle_3$ is the 1-dimensional harmonic oscillator transformation bracket for particles with mass ratio 3. This transformation bracket value is calculated as

$$\langle n_1' n_2' | n_1 n_2 \rangle_3 = \frac{\sqrt{n_1! n_2!}}{\sqrt{n_1'! n_2'!}} \sum_{k=0}^{n_1'} \binom{n_1'}{k} \binom{n_2'}{n_2 - k} \left(\frac{1}{2}\right)^{n_1' + n_2 - 2k} \left(\frac{\sqrt{3}}{2}\right)^{n_2' - n_2 + 2k} (-1)^{n_2 - k}. \qquad (2.19)$$

Diagonalizing $\hat{S}$ will give eigenvectors that are normalized linear combinations of product states with the same total harmonic oscillator number N. We again select the ones with eigenvalue 1 and keep those as our fully symmetrized 3-body basis. The project leading up to this thesis only worked with 2-particle and 3-particle systems and thus only the treatment of symmetrizing those bases is relevant to this work, the procedure is generalizable to symmetrize up to any A-particle basis. The details of this procedure are explained in [Reference Jurgenson].

2.2.4 Matrix Elements in the 3-Body Harmonic Oscillator Space

From equation (2.9), we are able to transform both parts of our 2-body Hamiltonian into the symmetrized 2-body harmonic oscillator basis. In order to work in a our

symmetrized 3-body basis, we need to compute the kinetic energy for the 3-body system and embed the 2-body potential in the 3-body space. Both the A-body kinetic energy and the A-body embedded 2-body potential are defined with respect to their $A-1$-body counterparts, so we will set up the discussion to give those formulas. Let $|N_A i_A\rangle$ be a symmetrized A-body state with total harmonic oscillator number N_A and degeneracy label i_A. $|N_A i_A\rangle$ is defined as

$$|N_A i_A\rangle = \sum_{l=1}^{k} c_l |N_{A-1} i_{A-1}; n_{A-1}\rangle \tag{2.20}$$

where k is the number of states in the linear combination, $|n_{A-1}\rangle$ is a harmonic oscillator state with respect to the $A-1$-th Jacobi coordinate, and $|N_{A-1} i_{A-1}\rangle$ is a symmetrized $A-1$-body state. The A-body kinetic energy is calculated as

$$
\begin{aligned}
\langle N_A' i_A' | \hat{T}_A | N_A i_A \rangle = \sum_{l=1}^{k} \sum_{l'=1}^{k'} c_l c_{l'}' (& \langle N_{A-1}' i_{A-1}' | \hat{T}_{A-1} | N_{A-1} i_{A-1} \rangle \delta_{n_{A-1}', n_{A-1}} \\
& + \frac{\omega}{4} \delta_{N_{A-1}', N_{A-1}} (\sqrt{(n_{A-1}+1)(n_{A-1}+2)} \delta_{n_{A-1}', n_{A-1}+2} \\
& + \sqrt{n_{A-1}(n_{A-1}-1)} \delta_{n_{A-1}', n_{A-1}-2} \\
& - (2n_{A-1}+1) \delta_{n_{A-1}', n_{A-1}})).
\end{aligned}
\tag{2.21}
$$

Similarly, the embedded 2-body potential in the A-body basis is calculated as

$$
\langle N_A' i_A' | \hat{V}_A | N_A i_A \rangle = \frac{A}{A-2} \sum_{l=1}^{k} \sum_{l'=1}^{k'} c_l c_{l'}' \langle N_{A-1}' i_{A-1}' | \hat{V}_{A-1} | N_{A-1} i_{A-1} \rangle \delta_{n_{A-1}', n_{A-1}}.
\tag{2.22}
$$

2.3 Similarity Renormalization Group

The similarity renormalization group (SRG), whose use in nuclear physics was initially explored at OSU, is one method of reducing the computational complexity of low-energy nuclear calculations. The idea behind it is to continuously unitarily transform the operator of interest (for example, the Hamiltonian) into a simpler form. This

simpler form is chosen to allow the large basis to be truncated without affecting the operator eigenvalues, which are essential to the truncated operator's utility in later calculations. The SRG transformation is given by the flow equation for the evolving Hamiltonian H_s,

$$\frac{d\hat{H}_s}{ds} = [\hat{\eta}_s, \hat{H}_s] \tag{2.23}$$

, where $[A, B]$ is the matrix commutator and $\hat{\eta}_s$ is the generator of the transformation which is defined as

$$\hat{\eta}_s = [\hat{G}_s, \hat{H}_s] \tag{2.24}$$

where $\hat{G}_s$ is known as the flow operator. The changes in the Hamiltonian over the course of the SRG evolution are absorbed into the potential, $\hat{V}_s$ leaving the kinetic energy, $\hat{T}$, independent of s, the flow parameter. s is taken to be 0 for the unevolved Hamiltonian. It is often convenient to use $\lambda = 1/s^(1/4)$ as an alternative flow parameter, so instead of going from $s = 0$ towards $s = \infty$ over the course of an SRG transformation, you are going from $\lambda = \infty$ towards $\lambda = 0$. For our purposes, $\lambda = 40$ will be a good enough place to start with the initial Hamiltonian.

2.3.1 Induced Many-Body Forces and Flow Operators

The SRG evolution is fully unitary for an A-body when done in the A-body space. However, working in the full many-body space is only feasible for the smallest of systems, due to the combinatorial scaling of the basis size with respect to A. When evolving an A-body system operator in some smaller basis, the SRG evolution induces many-body forces which show up as an error when eventually computing operator observables further down the line. This error has limited the SRG's domain of applicability to calculations for small to medium-sized systems.

Certain results have suggested that alternative choices for the flow operator, $\hat{G}_s$, could reduce this many-body force induction and thereby reduce the error induced

by applying SRG to calculations. The standard choice for $\hat{G}_s$ is the kinetic energy, which is easy to calculate and represent in most problems. Alternative flow operator choices have been explored previously, but not thoroughly.

2.3.2 Value of 1-Dimensional Laboratory

SRG was explored initially in the 1-dimensional setting, which made it easy to test and allowed those exploring it to learn a great deal, such as the more careful treatment necessary for A-body evolutions in momentum space. Moreover, the setup of the 1-dimensional system is analogous to that of more complex calculations, so the results from the 1-dimensional setting generalize to application of SRG in 3-dimensional calculations. If there is something to be learned about the connection between flow operator and many-body force induction, it is best comprehensively explored in a simple 1-dimensional problem and then generalized to expensive, difficult 3-dimensional calculations.

We will now explain the structure of the Python framework designed to make it easy to explore SRG in this 1-dimensional setting.

Chapter 3
SRG1D PYTHON LIBRARY

In this chapter, we discuss the design and features of the `srg1d` Python library. The development of this library was a significant portion of the work that led to this thesis. We will begin by reviewing some programming concepts and terms to facilitate the explanation of the library. We will also speak briefly about the Python programming language and the reasons it was chosen for this project. Then, we will move into the design of the library overall as well as the specific parts. Finally, we will discuss details regarding the implementation and testing the library.

3.1 Review of Programming Concepts

This review assumes the reader has a basic understanding of programming and aims to explain certain programming concepts and communicate their practical importance. This will better equip readers to understand the rest of the chapter.

- **Classes** are a way for programmers to create complex objects for cases where the programming language basic types (integers, floating point numbers, strings, booleans, arrays) are not enough to meet their needs. A class has some internal data, as well as methods and properties which interact with that internal data. A user can create an object of a class by calling its constructor method. It can interact with the object through its interface.

- **Interfaces** define how a user can interact with an object, through methods and properties. Methods typically change the object in some way or do some significant evaluation of the object internals. Properties allow users to extract certain properties of the object, without changing them.

- **Collections** are classes that contain a bunch of similar objects and define how users can access the objects in the collection. Different variations of collections exist to enforce some conditions on the collection, such as no duplicates or limited ways to access objects in the collection.

- **Indexing** is an operation defined on a collection which allows a user to get any object in the collection by providing the collection the proper index. The simplest example of indexing is a standard array, which most people who have used a programming language have seen.

- **Iterating** on a collection is a way for a user to sequentially access every object in a collection. For a user, accessing objects in a collection via iteration requires the user to know as little as possible about the collection or how to actually access the objects in the collection. Some collections guarantee that the order of the objects accessed via iteration is the same every time as long as the collection is not modified.

- **Mutability** is the ability of an object to be modified by the user after it is created. Mutable objects (or objects that can change by something the user does) give the user more flexibility in terms of possible functionality, but open the door for potential misuse by the user when properties of the collection are saved by the user, the collection is modified, and those saved properties are not updated to reflect the changes. Immutable objects make sense in cases where the objects should not change, like an array of the days of the week.

- **References** allow multiple objects to have access to the same thing. If the reference to an object is given to two different objects and the first modifies the referenced object, the changes will be reflected in the second object as well. This can lead to difficult to manage behavior. As a result, it is good to use references in conjunction with immutable objects, because in that case both objects can be sure that the object their reference refers to will never change.

3.2 A Note on the Python Programming Language

Python is a dynamically-typed, interpreted, high-level programming language that is used for general-purpose programming. Python is used by many, both inside and outside the sciences, and according to StackOverflow, it has "a solid claim to being the fastest growing major programming language." Python currently has two supported major versions, Python 2 and Python 3, with Python 2 being supported for developers with Python 2 codebases or rely on packages that have not yet transitioned to Python 3. Although the numerical libraries in Python are quite fast, thanks to their low-level C bindings, Python is still not a high-performance language. However, despite the general computational challenge faced by nuclear theory in general, the simple 1-dimensional system explored here is numerically relatively simple. Thus, high throughput and efficient memory usage are not requirements for these calculations to be done in a reasonable amount of time on any modern personal computer. In addition, Python is extremely expressive with a large standard library and well-documented third party libraries, making developer productivity very high. This was a priority in this research, leading to the decision to work in Python.

The next section discusses the general design of the library and references the interfaces for various different classes. A feature of Python is that users with sufficient knowledge of the internal representation of data in Python classes can directly

access those internals and use or modify them how they see fit. This means that published interfaces don't truly limit what a user can do with a library and things like immutability are difficult or impossible to truly enforce. However, Python developers have the philosophy that all Python users are "consenting adults." This means that the documentation of interfaces and conditions on the objects like immutability are communicated to users in an understanding that they should stick to these documented designs if they expect things to work as advertised.

3.3 Library Design

The class-based abstractions offered by the library offer consistent representations of different logical classes of objects present in any SRG calculation. By offering the user classes and functions that handle much of the complex, but frequently required functionality in any SRG calculation, the library allows the user to focus on exploration and research rather than reinventing the wheel. Behind the abstractions in the library are also several checks that ensure the physical correctness and consistency of what the user is doing. This further boosts user productivity, as the library provide transparent clear errors as to what conditions were not met, allowing the user to quickly understand where the error in their program is.

The library is roughly split into two logical halves. The data representation half (`state`, `basis`, and `operator`) handles the representations of the matrices for the operators in the calculations. Its primary goal is to add the context to the numerical representation of matrices and as a result make transformations of those matrices easier to achieve. The evolution half of the library (`srg`) handles the SRG transformation of the data. While the data representation classes correspond to concrete objects, the `srg` class is more of a state machine that allows the user to put some data in, turn the crank, and get out intermediate and final results.

3.3.1 `state`

The `state` module provides the fundamental building blocks for the data representation side of the library. It contains two classes, `ho_state_2body` and `ho_state_nbody`, that implement the same simple interface. The interface for both classes simply defines a constructor, and `val`, a method to evaluate the harmonic oscillator wavefunction that the state corresponds to at some momentum. These classes are intended to add context data (harmonic oscillator number, number of particles, dirac notation) to the wavefunction that will simplify the evaluation of the wavefunction and allow for the construction of states with more complex structures due to additional physical conditions imposed on the states, like symmetry. Because these states are only intended to be used to get wavefunction values, their interface defines them to be immutable.

`ho_state_2body` represents a 2-body harmonic oscillator state. `ho_state_nbody` is different from `ho_state_2body` in that it represents a general linear combination of n-body product states. The only conditions on this linear combination of product states are that it is normalized and that the total harmonic oscillator number for each product state in the linear combination is the same. `ho_state_2body` is the basic building block of these general n-body product states.

It is worth mentioning that `ho_state_nbody` does not enforce any symmetry on these states. The conditions imposed on the linear combination of product states are the minimum conditions for such a state to be physically useful. The burden of ensuring proper symmetry falls on the operations creating the states, which is handled elsewhere in the `srg1d` library. Thus, for the purposes of an SRG calculation, a user should never have to deal with the `state` module directly.

3.3.2 `basis`

The `basis` module provides a basic interface for a `basis` class and two classes, `p_basis` and `ho_basis`, that implement this interface. The `basis` class interface comprises only a constructor, but it also defines `__len__`, giving the size of the basis, and `__getitem__`, allowing the user to access a state by its index. These additionally defined methods make any basis indexable and iterable, with iterability being especially valuable for working with bases. The classes that implement the `basis` interface are also defined to be immutable, because after construction these bases have certain properties (symmetry, completeness, etc.) that would be disturbed by the removal, addition, or modification of a state in the basis.

`p_basis` is the representation of a momentum basis. Strictly speaking, a momentum basis is a continuous basis, that is operators need to be defined for any pair of real momenta. However, for numerical purposes, it makes sense to discretize the momentum space and impose some upper and lower cutoffs on momenta included in the basis. The upper and lower cutoffs on a discretized momentum basis are left to the user, who can judge what the range of relevant momenta is. The discretization scheme for the points between those upper and lower bounds is given by a Gaussian quadrature, which provides points p_i and weights w_i at which to evaluate the function. This provides us with the momentum basis for a single Jacobi momentum. From the single momentum basis, we can generate the two Jacobi momentum basis by taking the Cartesian product between two single momentum bases, with the weight assigned to to each new state being the product of the two weights corresponding to the states in the ordered pair. This approach can be used to generate a basis for any number of Jacobi momenta. The data representation of the n-body momentum basis is simply a list of these states, which are represented as n-tuples with $n-1$ Jacobi coordinates and finally a weight for the state.

`ho_basis` is the representation of a harmonic oscillator basis. This is simply a list of harmonic oscillator states (2-body or n-body) up to some maximum total harmonic oscillator number, N_{max}. The standard constructor for `ho_basis` generates a basis of 2-body states with proper symmetry. `ho_basis` has a second constructor `ho_basis.from_basis` that takes any n-body `ho_basis` object and generates an appropriately symmetrized $n+1$-body basis. Additionally, each n-body basis has a reference to the $n-1$-body basis used to generate it, which is useful for functions which need to recurse down to the 2-body basis. Because generating bases with appropriate symmetry is an expensive operation, it is important to avoid recreating `ho_basis` objects. Thus, the `srg1d` library makes an effort to work with references to existing objects whenever it is possible.

3.3.3 `operator`

The `operator` module provides a simple `operator` class which couples a matrix with a basis that corresponds to its representation along with several additional useful methods that work on operators. The packaging of a matrix and a basis into one object reduces the potential for bugs when calling methods that require both the data and the basis, which occurs frequently in the setup of an SRG calculation. It also ensures that methods requiring operators can be certain that the basis and matrix are consistent, so methods do not need to do redundant checks, such as ensuring the basis and data sizes match.

The methods provided by the `operator` module do one of three things: generate operators, transform operators, or embed operators. The operator generating methods, `create_ho_basis_kinetic_energy`, `create_p_basis_kinetic_energy`, and `create_kinetic_energy`, are methods that calculate the kinetic energy operator in a certain basis and return a corresponding `operator` object. While the general n-

body kinetic energy can be calculated directly, the n-body kinetic energy is calculated recursively, with the 2-body kinetic energy being the base case. This is where the n-body `ho_basis` internal reference to the $n-1$-body `ho_basis` comes in handy.

The `transform_operator_to_p_basis`, `transform_operator_to_ho_basis`, and `transform_operator` methods all compute a unitary transformation for the operator data between a harmonic oscillator basis representation and a momentum basis representation. The harmonic oscillator basis has a parameter ω which relates to the strength of the oscillator which needs to be optimized for the transformation from momentum space to harmonic oscillator space. If no value is given for ω in the function call, it will be optimized internally and return with the transformed operator. A transformation from harmonic oscillator space to momentum space requires the ω used originally to transform the operator to harmonic oscillator to be passed in by the user. A future implementation may add an internal member to the `operator` class keeping track of ω for operators in harmonic oscillator space to avoid placing the burden of keeping track of the value on the user.

Finally, `embed_operator` embeds an operator in harmonic oscillator space in a higher-body harmonic oscillator basis provided by the user. The logic for this embedding is similar to the logic for generating the general n-body kinetic energy in harmonic oscillator space. Again, this algorithm takes advantage of the internal reference to the $n-1$-body basis kept by the n-body `ho_basis` to avoid recreating a new equivalent basis.

3.3.4 srg

The `srg` module only contains the `srg` class, which handles the evolution of some operator via the SRG flow equation. The three methods of this class are its constructor, `evolve`, and `set_generator`. The constructor takes in a potential and a kinetic

energy (which are the Hamiltonian when added up), as well as a 2-tuple of matrices called the generator. These two generator matrices are weights assigned to matrix elements of the potential and kinetic energies that are used when computing the flow operator G_s. For the standard $G_s = T_{rel}$, a zero matrix for the generator corresponding to the potential energy and a matrix of ones for the generator corresponding to the kinetic energy is one way to compute the correct flow operator. Allowing the user to provide a function to compute the flow operator from the kinetic and potential energy was a design that was considered, but rejected in favor of the easier-to-check and less complex generator matrices. The `set_generator` method allows the user to alter the flow operator used between evolutions. While it is not typically useful to change flow operators part of the way through a full SRG calculation, this was added to allow us to easily switch between operators during calculations and investigate how composite SRG evolutions affect results.

The `evolve` method evolves the Hamiltonian provided in the constructor to a specified value of the flow parameter λ. The method uses `scipy.integrate.ode` to solve the system of coupled differential equations. In addition to taking the target value of λ as a parameter, it also optionally takes parameters to specify an integrator and parameters for that integrator for users with specific needs and familiar with the integrators available through the `scipy.integrate.ode` method. The default integrator used is ´dopri5´, a fourth order Runge-Kutta method that works well for non-stiff systems. However, for problems that are more stiff, it makes sense to switch to ´lsoda´, a solver that dynamically switches between different algorithms for non-stiff and stiff problems. The only downside to the ´lsoda´ integrator is that is not re-entrant meaning that different `srg` objects cannot both use it at the same time and therefore would be unable to run in parallel. The `srg` class also has accessor functions to allow the user to quickly access the data of the evolved potential as well

as the current value of the flow parameter *lambda*.

3.4 Implementation

The first version of the library was written to be compatible only with Python 2̇7. We made a decision fairly early on to focus development on Python 3, with an auxiliary of still being Python 2 compatible. Over time, the attention paid to keeping the code Python 2 compatible waned. As a result, it is currently not able to be used in Python 2. Fortunately, the recent Python Developers Survey 2017 published by Jetbrains shows that Python 3 is used by 75% of Python developers as their primary version of the language, a substantial shift from 2013, when this project began, and 2015, when the switch to Python 3 on the project was made. Additionally, Python 2 will officially stop being supported in 2020. All of this leaves the `srg1d` library in a good place considering Python 3 adoption trends and the waning support for Python 2.

3.4.1 Libraries Used

The only non-standard libraries used in `srg1d` are NumPy and SciPy. NumPy arrays are convenient for representing vectors and matrices and can be used interchangeably with lists and lists of lists. Additionally, they allow for efficient numerical calculations and are the standard representation for data throughout the NumPy and SciPy libraries. NumPy also offers matrix and vector operations. SciPy extends the utility of NumPy by offering more linear algebra functions such as eigenvalue decomposition and numerical solvers for systems of coupled differential equations. Both NumPy and SciPy are available in the Python Package Index (PyPI) and can be installed easily with the Python package management system, `pip`.

3.4.2 Handling Unexpected Usage

It was mentioned previously that Python relies on library implementers and library users being "consenting adults." As far as libraries are concerned, this means library implementers only have an obligation to document the interface, not actually enforce it. This is different from programming in languages like C++ or Java, where static typing forces users to adhere to an interface. In fact, Python users typically take the stance that if a library user is able to provide an object that doesn't strictly match the library interface, but interacts with the library implementation in a way that works, that usage is fine. This is known as "duck typing," following the principle "if it walks like a duck and talks like a duck, then it must be a duck." In keeping in line with this philosophy, the `srg1d` library never enforces the type of objects using `isinstance`. Instead, it typically checks that an object implements the necessary interface, which is done by checking the existence of the required methods and properties.

However, in cases where data is taken in but not used, like in an object constructor call, it makes sense to ensure that errors that will definitely lead to problems later are immediately detected and pointed out to the user. Examples of this include checking that matrices that will be multiplied later have the correct dimensions and checking that corresponding bases and matrices have the same length. The implementation of the `srg1d` library strives to do comprehensive checks in this regard wherever it is reasonable, primarily in object constructors.

One notable exception to this rule is the `val` method from the `ho_state_2body` and `ho_state_nbody` class. This method is ideally never directly called by the user, but profiling done with `cProfile` showed that around 75% of time spent initializing the SRG calculation (calculating operators and transforming or embedding them) was spent inside of it. Removing the checks done on the parameters in `val` halved the time spent inside it. Outer loop optimizations also reduced the number of calls to the

method from several million to several hundred thousand. As the library is designed to make it so the user never has to use this method and it is so frequently called, we believe removing error checking from it is worth the performance. In cases where the user does need to directly work with the `ho_state_2body` and `ho_state_nbody` objects, they still have the documentation of the interface to fall back on.

3.5 Testing

An effort was made to adopt a test-driven development (TDD) approach in the development of the `srg1d` library. Tests are written using the `unittest` standard Python library, and the documented way to run the tests involves using `nose`. `nose` is an extension to `unittest` that can easily be installed via `pip`. The installation of `nose` comes with a script called `nosetests`, which can be run by the user and will automatically discover and run tests. While the project does not have complete code coverage, the existing tests focus on object constructors and specifically on graceful failure when these constructors are incorrectly called. The `state` module has good test coverage of the `val` method for both classes. One of the goals for the library before publication of the package is to extend the current incomplete test suite to have better coverage.

Chapter 4
Approach to Exploration

4.1 Objectives

4.1.1 Verify Many-Body Force Induction

4.1.2 Test Alternative Flow Operators

4.2 2-Particle Systems

4.2.1 Verify Momentum Space Results

4.2.2 Preliminary Alternative Flow Operator Tests

4.2.3 Transform to Harmonic Oscillator Space

4.2.4 Check Kinetic Energies

4.3 3-Particle Systems

4.3.1 3-Particle Basis Symmetrization

4.3.2 Verify Computed 3-Body Eigenvalues

4.3.3 Test $\hat{G} = \hat{T}$ Many-Body Force Induction

4.3.4 Test Alternative Flow Operator Many-Body Force Induction

Chapter 5
RESULTS

5.1 2-Body

5.1.1 Momentum Space SRG

5.1.2 Harmonic Oscillator Space Transformation

5.2 3-Body

5.2.1 3-Particle Ground State

5.2.2 2-Body vs 3-Body SRG Evolution

5.2.3 Alternative Flow Operator Tests

Chapter 6
Conclusion and Outlook

6.1 Future Development on `srg1d`

6.2 Conclusion

6.3 Open Questions and Outlook