The Similarity Renormalization Group

Theoretical & Numerical aspects of Nuclear Physics: project

Giuseppe Ventura

March 2023

Contents

1 Introduction

The search and study for a Nuclear potential, capable of correctly describe nuclear interactions in the nuclei, has been one of the biggest challenges of the modern theoretical nuclear physics.
While methods to derive it starting from the more fundamental nuclear interactions (based on lattice QCD) or from an EFT description of the nuclear phenomena are in development, the abundance of experimental datas allowed, in the last decades, to build a potential based on the known symmetries and made so that it reproduces the observations by tuning a wide number of free parameters.
The latter class is referred to as the "**realistic**" potentials, among them it is possible to cite ArgonneV18 and CD-Bonn, used in nuclear matter structure applications.
However, implementing the realistic potentials has its own challanges, in fact they come with the problem of the presence of a strong repulsive core in the short range ($r \leq 1$ fm), which is so repulsive that some of the first models would consider it as an infinite barrier. By studying the potential in the momentum space this can be related to the strong coupling existing between the large and small momentum regime, turning out as strong off-diagonal components of the matrix elements of the potential.
One possible solution to this problem is to find a method to "pre-diagonalize" the potential so that its application to more difficult environment has better performance. This has to be done without changing the physics of the problem, and therefore without changing the eigenvalue of the Hamiltonian.
The (Similarity Renormalization Group) SRG Method is suitable for this task where a continuous number of unitary transformations is applied to the Hamiltonian and

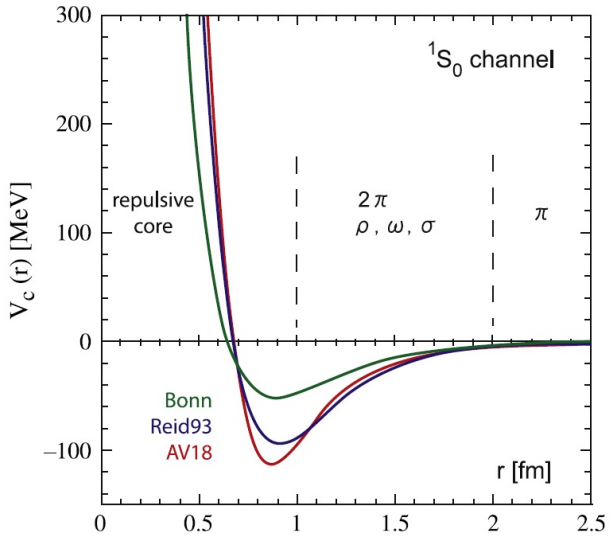

Figure 1: Schematic representation of some realistic potentials in the position space for the $^1S_0$ channel

the flow equation is made suitable for the pre-diagonalization. As we will see in the next sections, the main advantage of the SRG is that it produces interactions that are softer than the original ones, which allows for more efficient calculations in many-body systems. However, the SRG is not free of problems, and several issues need to be carefully addressed, such as the choice of the initial interaction, the selection of the SRG generator, and the truncation of the evolved interaction.

The SRG evolution has potential applications in the context of many-body systems, where the nucleon-nucleon (NN) potential is combined with three-nucleon (3N) and higher-order forces to model a more complex system. In such environments, several "ab-initio" methods have been developed to solve the problem, with the No-Core Shell Model being one of the most widely used. To solve the Schrödinger equation in this model, an eigenvalue problem must be solved. However, the matrices involved in this problem increase factorially in size with the nucleon number, making the evaluation of the problem more complex.

In this context, the application of the SRG evolution can become crucial in order to soften the matrix and simplify the evaluation of the problem. By applying the SRG

evolution, the matrix can be transformed into a simpler form, making it easier to solve the eigenvalue problem.
The report is structured into three main sections. **Section 2** presents an overview of the code's structure, outlining its primary functionalities and examples of implemented codes. In **Section 3**, we investigate the application of the SRG to a realistic potential. Lastly, in **Section 4**, we examine the application of the SRG evolution to an Hamiltonian constructed from a three-nucleon system, specifically the triton, studying the evolution of the matrix elements as well as the ground state of the system.

1.1 Derivation of the flow equation

The starting point is:

$$H(s) = U(s)HU^{\dagger}(s). \tag{1}$$

By deriving w.r.t. the flow parameter s I get:

$$\frac{\mathrm{d}H(s)}{\mathrm{d}s} = \frac{\mathrm{d}U(s)}{\mathrm{d}s}HU^{\dagger}(s) + U(s)H\frac{\mathrm{d}U^{\dagger}(s)}{\mathrm{d}s}. \tag{2}$$

By deriving the unitarity constrain $U(s)U^{\dagger}(s) = 1$ we can simplify the relation to:

$$\frac{\mathrm{d}H(s)}{\mathrm{d}s} = [\eta(s), H(s)], \tag{3}$$

with $\eta(s) = \frac{\mathrm{d}U(s)}{\mathrm{d}s}U^{\dagger}(s)$ an anti-hermitian operator. The $\eta(s)$ operator choice will characterize the behaviour of the evolution, an usual choice is the following:

$$\eta(s) = [T, H(s)] \tag{4}$$

The motivation for this choice is related to the known fact that the differential equation will be driven to the fixed point s.t. $\mathrm{d}H(s)/\mathrm{d}s = 0$, this will happen, except for the trivial case, when the Hamiltonian is diagonal in the kinetic energy eigenstates, a.k.a. the momentum space.

2 The code

The code to solve the illustrated problem has been completely written from scratch and it is available in the dedicated github page.
The idea of the program is to solve the differential equation where all the actions can be done in the same program and all the inputs, as well as the analysis of the results, can be controlled by the user.

2.1 Structure of the program

The program is initiated upon compiling the "main.py" file, which presents the user with two options: entering the SRG Solver environment or plotting the given initial data, namely the kinetic and potential matrices. Upon entering the Solver environment, the user can choose from a selection of numerical algorithms to solve the RGE equation. After solving the equation, the user can then enter the analysis environment where they can study the behavior of the solution. One method of analysis involves examining the eigenvalue problem, which allows for the study of how the ground state evolves with respect to the evolution parameter s. As mentioned in the previous section, it is expected that the ground state will vary very little, providing an indication of the accuracy of the evolution. Additionally, the user can plot and save the obtained results, such as the evolved potential using a heatmap for different values of the evolution parameter. Finally, the evolved potentials can be written to file for future reference.
A schematic visualization of the program structure is provided by the figure (2).

2.2 Classes

In order to organize the large number of functions present in the program, all the functions are grouped into Classes. Although in this particular program, these classes don't serve the purpose of Object Oriented Programming, they are still useful to organize the functions in a logical and efficient way. The classes that make up the code are as follows:

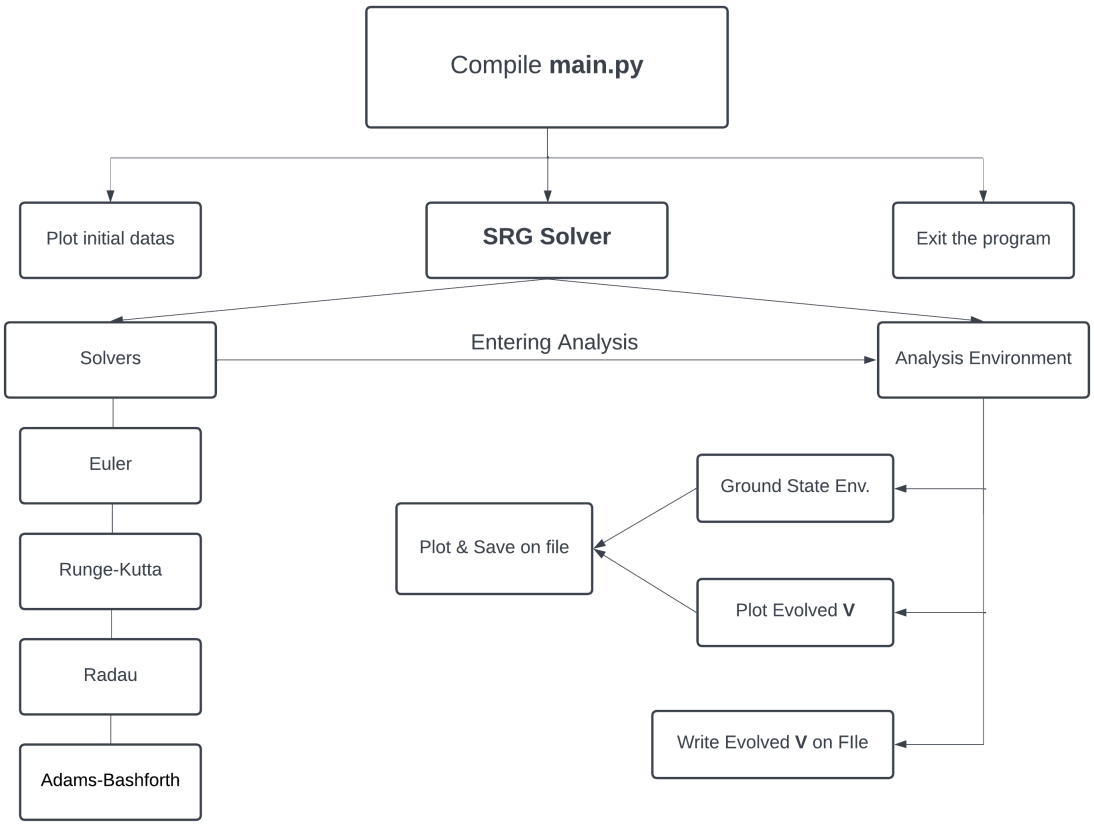

Figure 2: The program's functionalities are illustrated in this schematic, demonstrating the diverse range of tasks that can be accomplished through its use.

- **Constants**: This class groups together all the constants, such as $\hbar$ or the initial matrices, as the name suggests.
- **Solvers**: This class contains all the routines used to solve the RGE equation and is analyzed in depth in the next section. The solvers include various numerical methods such as the fourth-order Runge-Kutta method and the Adams-Bashforth method, which are used to solve the differential equations that arise during the RGE evolution.
- **Checks**: This class contains functions that are called during the program

execution to ensure that the user input is compatible with the environment in which we are working. For example, the *CheckIntPositive* function checks that the number of ground states we want to compute is a positive integer that does not exceed the number of integration steps. These functions are simple yet crucial for the proper execution of the program.

- **Plots**: As the name suggests, this class contains all the plotting routines used in the program. These routines allow for the generation of a variety of plots, including plots of the initial matrix elements, plots of the evolved potential for different values of the s parameter, and plots of the ground state as a function of s. The user can choose to save the plots to disk, and the process of doing so is automated. The program creates folders and files as needed to ensure that the plots are stored in an organized and accessible manner.

- **GroundStateCompute**: This class is responsible for computing the ground states of the system for a selected or all values of the evolution parameter. The class contains a single function that carries out the computation of the ground states and stores them in an array that is utilized by the plot routines.

- **Switchers**: This class contains functions that allow the user to switch between different program environments by making choices, such as selecting a specific routine for a given task. These functions are designed to link the user's choice to the appropriate routine within the program, allowing for smooth transitions between different modes of operation. It serves as a useful tool for ensuring that the program remains organized and easy to use.

- **WriteOnFile**: This class is responsible for handling all writing operations to files. It contains several functions that can write data to a file, including the evolved matrix elements and the ground states. The functions in this class also handle the organization and management of files and directories for storing data. When writing data to a file, the class creates new directories and files if necessary and ensures that the data is stored in the correct format.

2.3 The SRG Solver

The SRG solver environment is one of the key components of the program, as it provides users with the ability to solve the SRG evolution equation using a choice of four different numerical methods. Of these methods, the "Euler" method at first order and the "Gunge-Kutta" method at 4-th order have been implemented manually in the code, while the "Radau" and "Adams-Bashforth" algorithms have been incorporated as routines within the Scipy package. These methods have been adapted to work with the specific problem at hand by, for example, reshaping the matrix object into a one-dimensional array for use in the evolution process.

In addition to providing a means for solving the evolution equation, the SRG solver environment also allows users to access directly the Analysis environment, which is described in detail below. This implementation enables secondary analysis of an already-evolved Hamiltonian, without the need to re-solve the evolution equation. However, it is worth noting that a check is in place to ensure that a solution is actually defined before proceeding with the analysis, as such a check is necessary to avoid errors arising from the absence of an evolved Hamiltonian.

All the numerical methods implemented in the program for solving the RGEs have been designed to output the evolved Hamiltonian and, therefore, the evolved potential in the form of an array. The ordering of the elements in the array follows the integration step chosen by the user at the beginning of the program. This feature allows for easy storage and subsequent analysis of the evolved potential at different values of the evolution parameter. Additionally, the use of an array ensures that the evolved potential is easily accessible for further computations or plotting, making it a valuable tool for investigating the behavior of the system under the influence of the SRG evolution.

An excerpt from the code pertaining to the **Solvers** class is presented in fig. (3), showcasing the manually implemented numerical algorithms for solving the RGE equation.

```
def Euler(Del, n):
    s=0
    print("Initializing the evolution...")
    HMat_s= []
    VMat_s= []
    HMat = np.copy(Constants.HMat0)
    HMat_s.append(HMat)
    VMat_s.append(Constants.VMat)
    for i in range(n):
        HMatDel = Del * Solvers.RightHand_2D(s, HMat)
        HMat = HMat + HMatDel
        s = round(s + round(Del,5),5)
        print("\033[F\033[K", end="")
        print("Successfully evolved up to s = ", s,"/",round(n*Del,5))
        HMat_s.append(HMat)
        VMat_s.append(HMat-Constants.TMat)
    print("\033[F\033[K", end="")
    print("Done!")
    return HMat_s, VMat_s

def RungeKutta4(s, HMat, ds):
    k1 = ds * Solvers.RightHand_2D(s, HMat)
    k2 = ds * Solvers.RightHand_2D(s + ds/2, HMat + k1/2)
    k3 = ds * Solvers.RightHand_2D(s + ds/2, HMat + k2/2)
    k4 = ds * Solvers.RightHand_2D(s + ds, HMat + k3)
    return HMat + (k1 + 2*k2 + 2*k3 + k4) / 6

def RKEvolution(ds, s_f):
    print("Initializing the evolution...")
    HMat_s=[]
    VMat_s=[]
    HMat= Constants.HMat0
    HMat_s.append(HMat)
    VMat_s.append(Constants.VMat)
    s=0
    while s < s_f:
        HMat = Solvers.RungeKutta4(s, HMat, ds)
        s = s + round(ds,5)
        s = round(s, 5)
        print("\033[F\033[K", end="")
        print("Successfully evolved up to s = ", s,"/",s_f)
        HMat_s.append(HMat)
        VMat_s.append(HMat-Constants.TMat)
    print("\033[F\033[K", end="")
    print("Done!")
    return HMat_s, VMat_s
```

Figure 3: Implementation of numerical solvers for the SRG evolution. The manually implemented algorithm, namely the first-order "Euler" method and the fourth-order "Gunge-Kutta" method, are shown.

2.4 The Analysis environment

Upon defining a solution variable in the code, which requires at least one successful execution of the SRG solver, the user gains access to the Analysis environment. This environment is designed to facilitate switching between specific sub-environments, including the Ground State, Plot, and Write sub-environments.

In the Ground State sub-environment, the user can study the ground state by solving the eigenvalue problem and taking the minimum value over a certain range of evolution parameters. The number of ground states to be computed can be specified by the user, and they are automatically distributed across the entire range of evolution parameters. This approach saves significant computation time, as the eigenvalue problem does not have to be solved for every integration step, which can be a huge number.

The resulting ground state evolution as a function of the evolution parameter is stored in an array, which can be plotted against the evolution parameter itself or saved to disk. The program manages storage automatically, creating suitable sub-folders in the parent directory and generating appropriately named files as necessary.

The Plot sub-environment allows the user to plot the matrix elements for different number of the s flow parameter, the number of generated plot can be controlled by the user.

The generated figures can be saved in ".png" format, and, again, everything is automatically organized in sub-folders.

The Write sub-environment, instead, serves the purpose of write the evolved potential on a ".dat" file so that it can be used and plot with an arbitrary program such as Mathematica.

Moving freely through these environments allows the user to have complete control on the analysis of the evolution and it is possible to perform multiple studying without having to re-run the evolution. In the same program session different methods can be used to solve the SRG at different levels of accuracy as much as the computer computation power allows and completely controlled by the user.

As an example, the code that computes the ground state is shown in fig. (4).

```
class GroundStateCompute:

    # A function that computes the ground state for the evolved potential, we can \
    # choose how many do we want to compute, they will automatically distributed so\
    # that they cover the full evolution

    def Groundstate(GroundChoice, HMatEV, n_step):
        Eigvalues = []
        Ground_state_s = []
        if GroundChoice == 1:
            Bool = False
            while Bool == False:
                NGroundState = input("How many groundstates do you want to compute? :")
                Bool = Checks.checkIntPositive(NGroundState, n_step)
            NGroundState = int(NGroundState)
            k = n_step / NGroundState
            k = round(k)
        if GroundChoice == 2:
            NGroundState = n_step
            k = 1
        print("Computing {} ground states...".format(NGroundState))
        i = 0
        j=0
        while i <= n_step:
            if len(Ground_state_s) <= NGroundState+1:
                Eigvalues.append(np.linalg.eigvals(HMatEV[i]))
                Ground_state_s.append(np.min(Eigvalues[j]))
            i = i + k
            j = j + 1
        print("\033[F\033[K", end="")
        print("Done!")
        return Ground_state_s, NGroundState, k
```

Figure 4: Routine that computes a user-chosen number of ground states as a function of the evolution parameters.

In the next section we study the applications of the code to some given potentials, the different methods are compared and the results of the evolution are shown.

3 SRG Evolution for a realistic Nucleon-Nucleon potential

The nucleon-nucleon system can be studied in the center of mass (CM) frame, where the two-body system can be factorized into the relative degrees of freedom between the nucleons (referred to as internal) and the CM degree of freedom, which can be removed by rescaling the energy spectrum. In this context, the relevant Hamiltonian takes a specific form.

$$H = T_{\text{rel}} + V_{NN}, \tag{5}$$

where the relative kinetic energy is defined as $T_{\text{rel}} = q^2/2\mu$, where q is the relative momentum and $\mu = m/2$ is the reduced mass of the system. As mentioned in the introduction, one of the main difficulties in studying the nucleon-nucleon system using realistic potentials arises from the strong repulsive core at short ranges. This leads to computational challenges when solving the eigenvalue problem for the Hamiltonian. The SRG has emerged as a powerful tool to address this issue by suppressing the off-diagonal matrix elements. By transforming the Hamiltonian through the SRG evolution, the strongly repulsive core can be softened, leading to a simpler and more efficient numerical evaluation of the eigenvalue problem. This makes it possible to study the nucleon-nucleon system using realistic potentials and to obtain accurate and reliable results.

3.1 SRG equation: analytic approximation

As discussed in the first section, in order to implement the RGE evolution for a given system, it is necessary to determine the appropriate generator η for the evolution. In our specific case, we consider the following generator:

$$\eta(s) = [T_{\text{rel}}, H(s)]. \tag{6}$$

We have selected the relative momentum space as the basis to investigate the problem, which is spanned by the set of basis states $|q\rangle$ at definite momentum. To

facilitate the analysis, we perform a parameter redefinition of the flux equation by introducing the variable λ, which is defined as $s^{-1/4}$ and has units of inverse femtometers (fm^{-1}). The resulting equation for the SRG evolution can be expressed as follows:

$$\frac{\mathrm{d}V(\lambda)}{d\lambda} = -\frac{4}{\lambda^5}[[T_{\mathrm{rel}}, H(\lambda)], H(\lambda)] \tag{7}$$

In this scenario, the evolution is carried out by reducing the value of the evolution parameter.

The projection of both sides of this equation onto the momentum space, and the use of a basis comprising eigenstates of the kinetic operator allows the equation to be expressed as an integral equation.

$$-\frac{\lambda^5}{4}\frac{\mathrm{d}V(q,q',\lambda)}{\mathrm{d}\lambda} = -(q^2-q'^2)^2 V(q,q',\lambda) + \int_0^\infty \mathrm{d}p p^2 (q^2+q'^2-2p^2) V(q,p) V(p,q') \tag{8}$$

We now direct our attention to the off-diagonal matrix elements that are far away from the diagonal, i.e., when $|q-q'| >> 1$. In this scenario, the first term on the right-hand side of the equation dominates over the second term. Under this assumption, the differential equation can be solved analytically, yielding the following solution:

$$V(q,q',\lambda) = V(q,q')\Big|_{\lambda=\infty} \exp\left\{-\frac{(q^2-q'^2)^2}{\lambda^4}\right\}, \tag{9}$$

leading to a progressive suppression of the off-diagonal terms, as we expected.

3.2 The evolution

In this section, I show the content of my Bachelor's thesis project which involved studying the SRG evolution for a realistic Nucleon-Nucleon potential. The input used is the CD-Bonn potential provided by my thesis supervisor.

The code used in the project was based on the reference code [[1]] and not my own work, therefore I've chosen not to include it as a content of my project. Indeed, the code I developed is designed specifically for the Triton application, but the SRG application to the NN potential was included in this report to illustrate the power

of the method.

The evolution was performed for decreasing values of the evolution parameter, and the computational effort required increased as the parameter approached zero. The final value chosen for the parameter was $\lambda = 1.5$ fm^{-1}, which provided a stable and reliable result. A 3D comparison between the initial input potential and the final evolved potential at $\lambda = 1.5$ fm^{-1} is displayed in fig. (5), while the full evolution is shown in fig. (6).

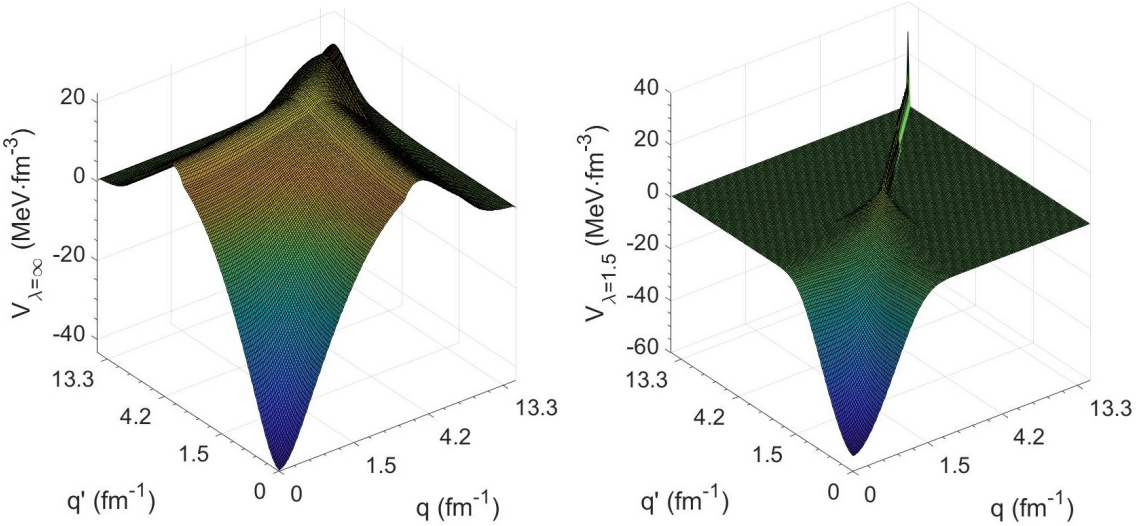

Figure 5: 3D comparison between the initial potential matrix and the full evolved one.

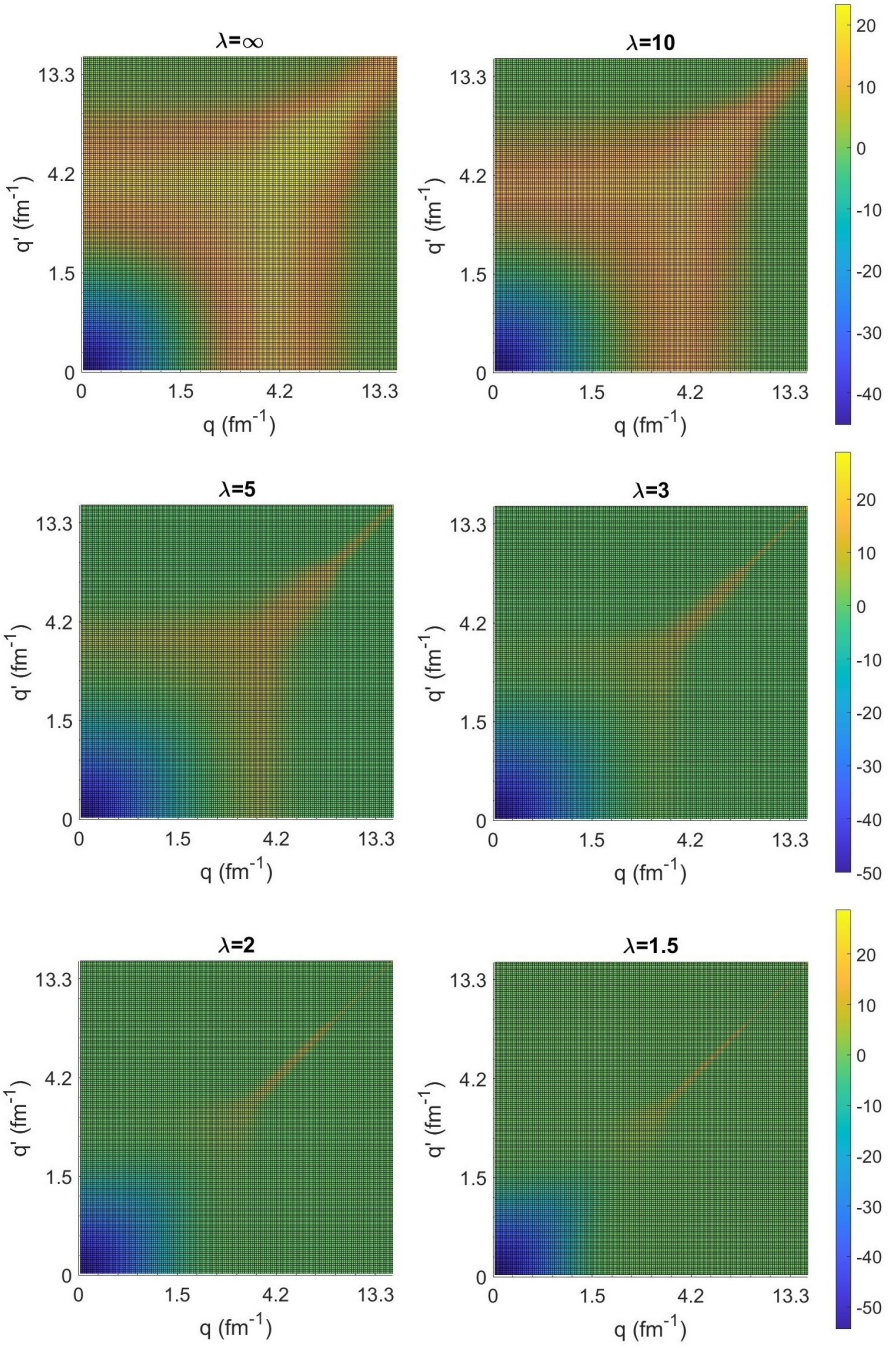

Figure 6: SRG evolution of the realistic CD-Bonn potential for decreasing value of the evolution parameter λ

4 SRG Evolution for the Triton ^3H chiral Hamiltonian in the No-Core Shell Model

In order to apply the Similarity Renormalization Group evolution to a given physical problem, we need to express the Hamiltonian operator, and therefore the kinetic and potential operators, in a given basis, so that it is possible to store and handle the datas in matrices that can be evolved through the differential equation.
In this section we study the evolution for the Triton Hamiltonian. The triton is a three-nucleon system and the basis chosen for this application is the antisymmetrized three-body Jacobi-coordinate harmonic oscillator basis.

4.1 The No-Core Shell Model

The No-Core Shell Model (NCSM) is a powerful method to treat quantum many body systems. The idea is to consider the nucleons "active" in contrast to what is assumed in standard shell-models.
It is implemented by treating the Schrodinger equation as a large scale matrix eigenvalue problem, considering the eigenstates expanded in an orthonormal basis of A-body states. The problem related to this approach is conceptually simple: the dimension of the Hilbert space is in principle infinite and so, to make the problem numerically tractable, a truncation is necessary [2].
Truncating the space leads to uncertaintes in the computation, making the solution to the eigenvalue problem not the exact one for the Schrodinger equation.
The truncation, still, is systematically performed and the only limitation is the computational power of our machine.
We can assign to each A-body configuration $|\Phi_i\rangle$ the energy quantum number corresponding to the sum of the single-particle energies that compose the configuration.

$$e(\Phi_i) = \sum_{\text{p in i-th conf.}} e_p \tag{10}$$

Among all the energies we can identify the configuration that minimizes it, also called "base determinant".
At this point we define the number of excitation quanta:

$$N(\Phi_i) = e(\Phi_i) - e(\Phi_{\min}) \tag{11}$$

At this point, the truncation is made by identifying a value $N_{\max}$ so that the set of the basis eigenstates can be expressed as:

$$B = \{|\Phi_i\rangle : N(\Phi_i) \leq N_{\max}\} \tag{12}$$

which makes the Hilbert space dimensionally finite. The number of basis states can be further reduced using the symmetry of the nuclear Hamiltonian, for instance the rotational invariance together with the parity. We can restrict the quantum number associated with these simmetries to have some fixed value. Multiple basis choices are possible to treat the Hamiltonian, but conventionally an Harmonic Oscillator one is chosen. The main advantages are related to the computational efficency and, as we see in the next section, the factorization of the center of mass states with respect to the intrinsic ones so that we can get rid of the CM excitations by just shifting the energy spectrum.

4.2 The three-body harmonic oscillator basis

The basis used to represent the triton Hamiltonian matrix elements is obtained by finding a basis of eigenstates of the following Hamiltonian:

$$H = \sum_i^3 \frac{\mathbf{p}_i^2}{2m} + \frac{1}{2}K(\mathbf{r}_{12}^2 + \mathbf{r}_{13}^2 + \mathbf{r}_{23}^2), \qquad \mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j, \tag{13}$$

which can be simplified by using the Jacobi coordinates:

$$\rho = \mathbf{r}_2 - \mathbf{r}_1, \quad \lambda = \frac{1}{\sqrt{3}}(2\mathbf{r}_3 - \mathbf{r}_2 - \mathbf{r}_1), \quad \mathbf{R} = \frac{1}{3}(\mathbf{r}_1 + \mathbf{r}_2 + \mathbf{r}_3). \tag{14}$$

From the coordinates it is possible to derive the conjugate momenta which we will respectevly labeled as $\mathbf{p}_\rho$, $\mathbf{p}_\lambda$, $\mathbf{P}$

$$H = \frac{\mathbf{P}^2}{6m} + \frac{\mathbf{p}_\rho^2}{m} + \frac{\mathbf{p}_\lambda^2}{m} + \frac{3}{4}K(\rho^2 + \lambda^2) \tag{15}$$

If we move to the center of mass frame we can remove the c.o.m. momentum, so that the system can be described by two 3-D harmonic oscillators. The basis can be expressed as:

$$|n_\rho, l_\rho, m_\rho; n_\lambda, l_\lambda, m_\lambda\rangle \tag{16}$$

We can then move to a basis with definite $l = l_\rho + l_\lambda$ angular momentum using CG coefficients.
The next step is to properly anti-symmetrize the basis so that the Pauli principle is taken into account for a system of three nucleon.
In this framework we can express the kinetic and the potential operators that from now on are considered given by the problem.

4.3 Initial matrix elements

The matrix elements has been uploaded in a python matrix using the following function from *numpy* python package:

```
T_Mat = np.loadtext('Kinetic_filename.dat')
```

The same has been done for the potential matrix so that *matplotlib* could plot them using a heatmap, the resulting plots are shown in figure (7).
We can immediately see that the potential is characterized by strong off-diagonal elements that we want to suppress using the SRG evolution equation.

4.4 The evolution

The code has performed the SRG evolution, and the resulting evolved Hamiltonian has been stored in a 2D array for each integration step, forming a 3D array where

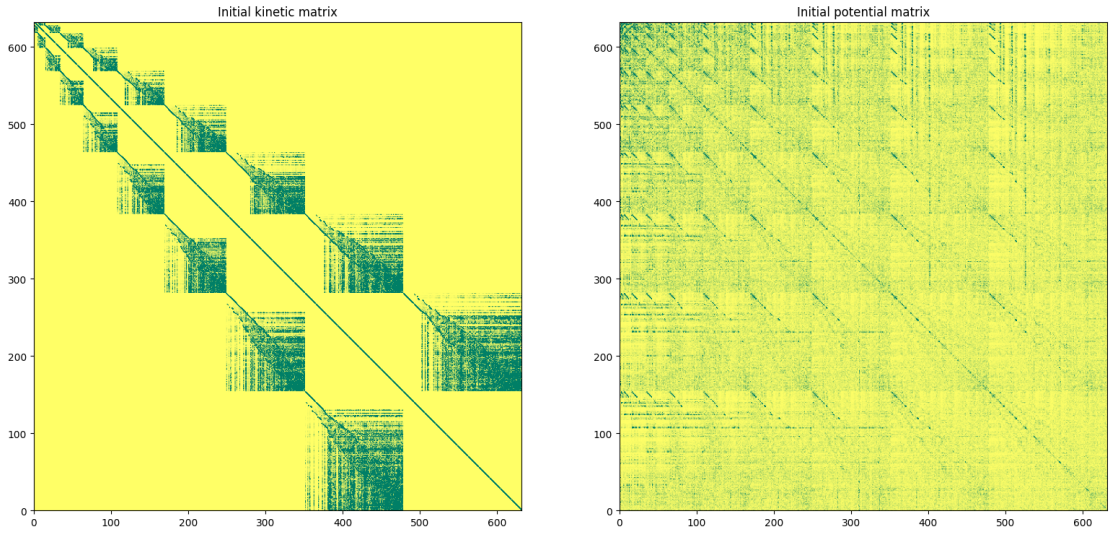

Figure 7: On the left the Initial kinetic matrix elements, on the right the potential

the first dimension corresponds to the running of s. The evolved potentials can be obtained by subtracting the assumed constant kinetic matrix from the evolved Hamiltonian.

Analysis of the results shows that the off-diagonal elements are correctly suppressed, and the main evolution occurs in the range of $s = [0, 0.5]$ MeV^{-1}. Beyond the latter, the evolution of the potential is nearly saturated, and no further changes are observed. This behavior is consistent across all the available algorithms in the code, suggesting that it is a characteristic of the differential equation operating on our datas.

The plots of the full evolution are displayed in the figures (8-10).

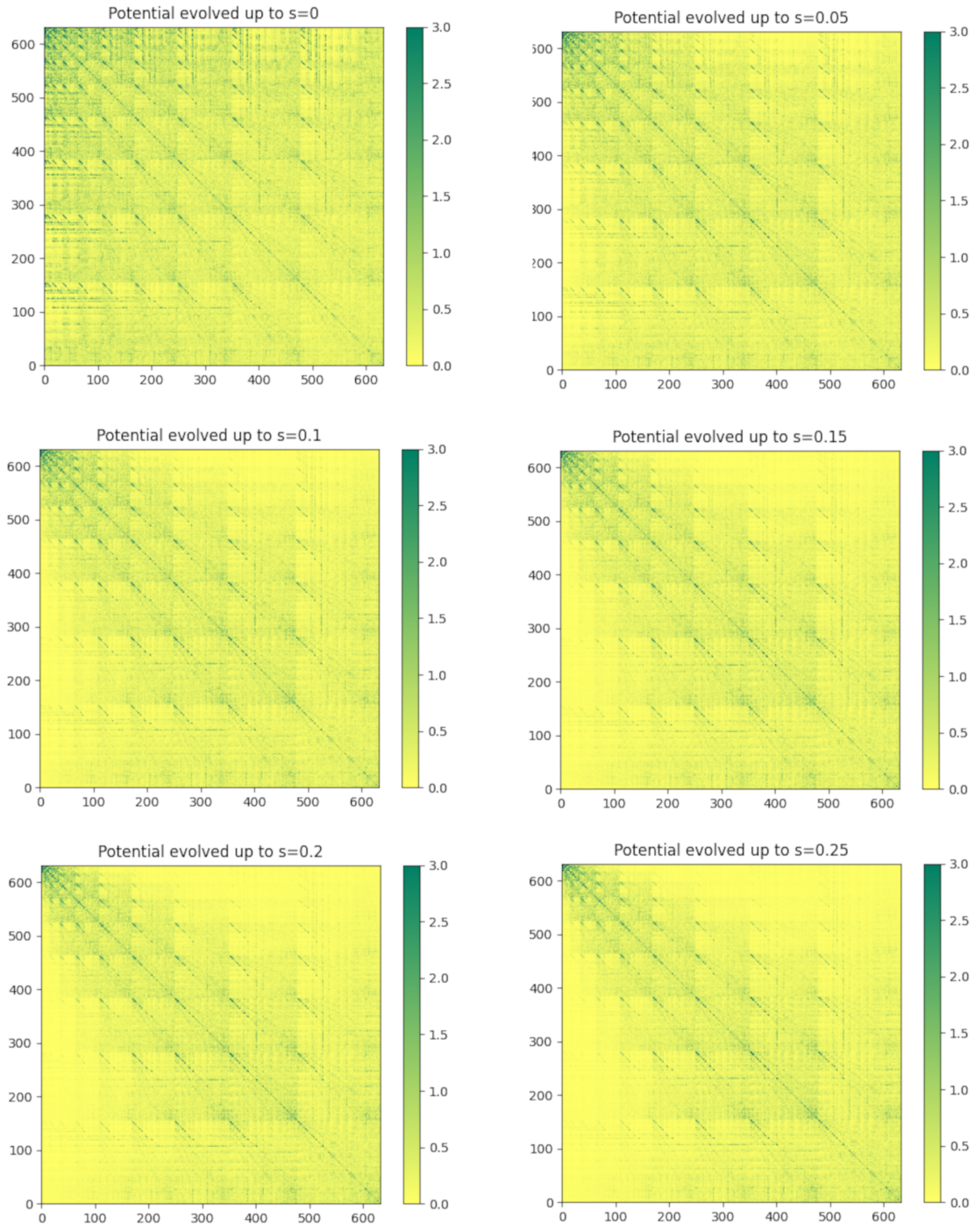

Figure 8: SRG evolution of the potential up to $s = 0.25$ MeV^{-1}

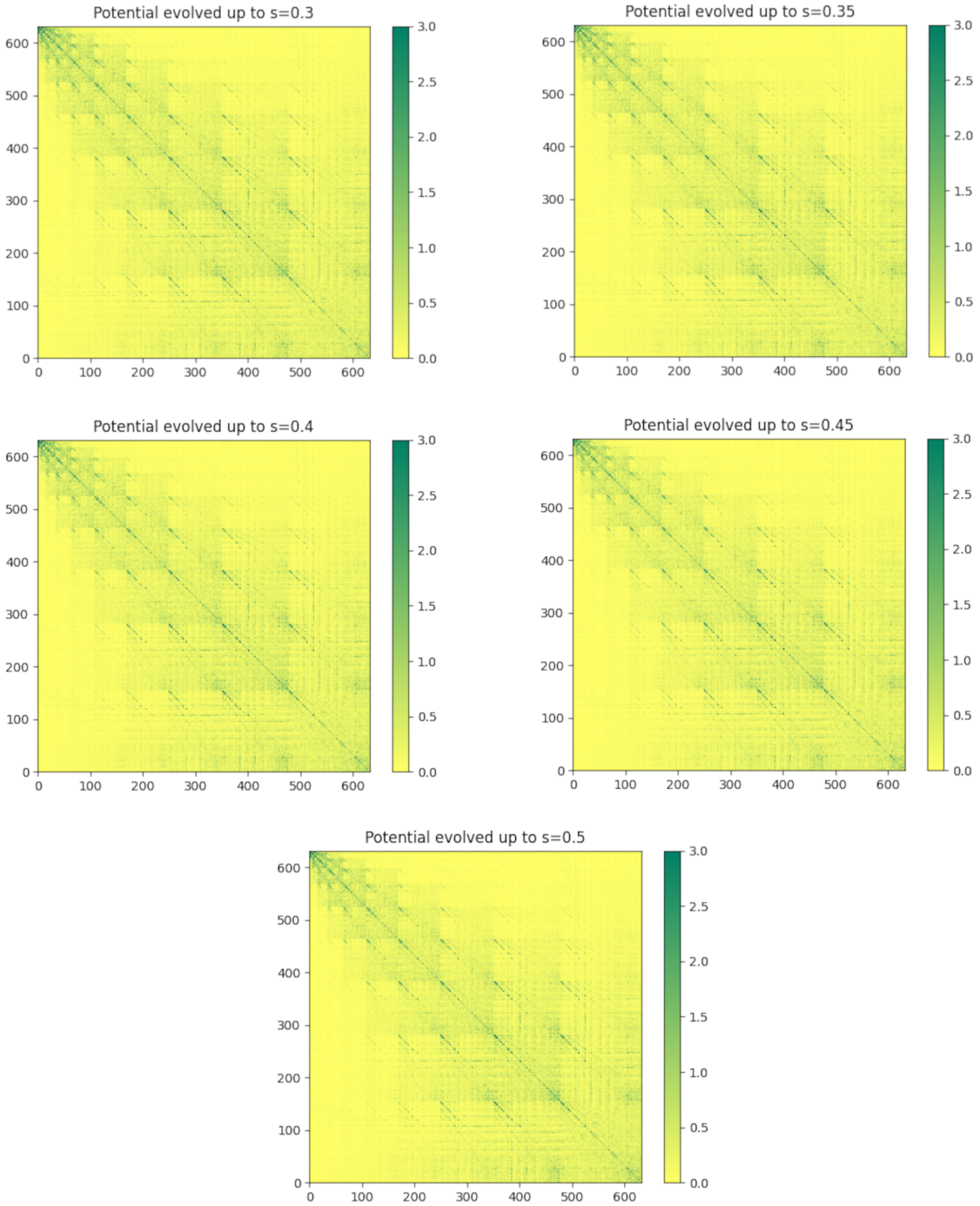

Figure 9: SRG evolution of the potential up to $s = 0.5$ MeV^{-1}

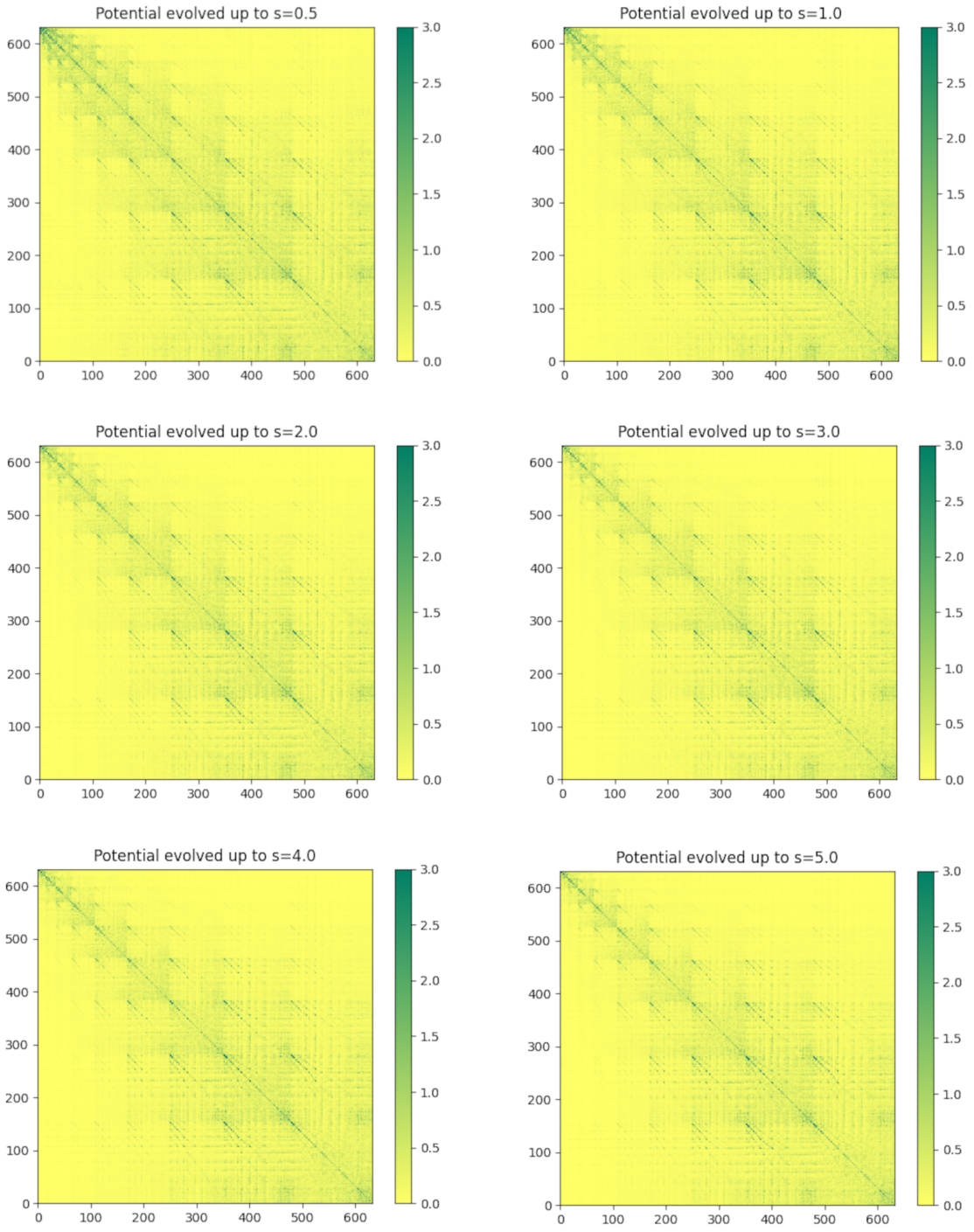

Figure 10: SRG evolution of the potential up to $s = 5$ MeV^{-1}

4.5 The ground state of the system

In this section, we investigate the solution of the Schrodinger equation via a matrix eigenvalue problem, which corresponds to a No-Core Shell method calculation. The implementation of this procedure is a simple matter of computing the eigenvalues of the Hamiltonian, which is automated in the *numpy* package with the function *linalg.eigvals(Matrix)*. The ground state of the system can be analyzed for different values of the parameter s, and we expect the ground state not to change significantly during the evolution due to the unitary transformations that leave the eigenvalues invariant. However, the evolution is not perfect as it is handled by a computer and is thus discretized, which can break the exact unitarity and lead to differences in the diagonalization process. To ensure an accurate SRG evolution, we require these differences to be minimized, and studying the evolution of the ground state provides insight into the algorithm's performance. From the structure of the code, we can infer the expected accuracy of the evolution methods. Specifically, the Euler method is anticipated to have the least accuracy due to its conceptual simplicity as a first-order approximation, whereas the Runge-Kutta method, being a fourth-order algorithm, is expected to perform more robustly.

Of the available algorithms implemented in the code, we anticipate that the *OdeInt* function from the *scipy* package will yield the most accurate results. This is largely due to the fact that it is a professionally designed algorithm, and that the integration steps specified by the user are actually considered as checkpoints for the evolution rather than true integration steps that are handled internally. As such, this method is likely to outperform other algorithms, given its superior accuracy and efficient design.

In considering the efficiency of an algorithm we also need to take into account the time taken by the algorithm to solve the problem at a given accuracy.

Several plots are displayed in the following where most of the studies are performed for $s = [0, 0.5]$ MeV^{-1}.

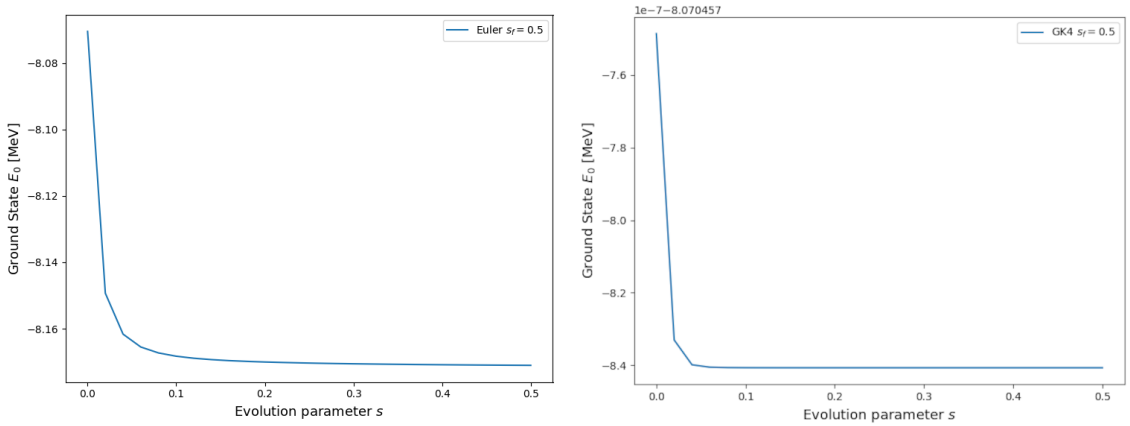

Figure 11: Ground state evolution for the Euler method (on the left) and Gunge-Kutta 4 (on the right) for $N_{\text{step}} = 1000$ and $s_{\text{final}} = 0.5$ MeV^{-1}

.

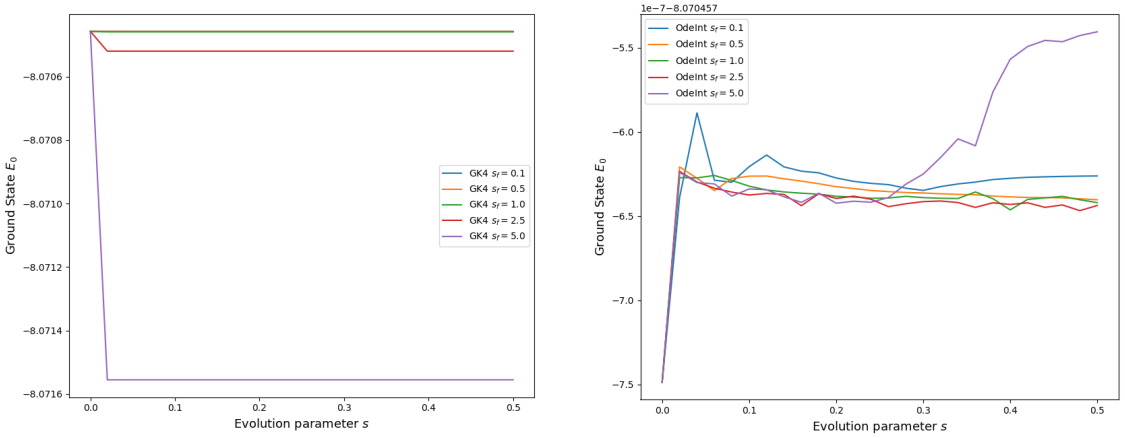

Figure 12: Ground state evolution for the Gunge-Kutta 4 and the Scipy OdeInt routine for different values of s_{final} and fixed $N_{\text{step}} = 1000$.

4.6 Algorithms compared

Based on the data presented, we can draw conclusions about the accuracy of various numerical methods for solving a given differential equation. Firstly, we compared

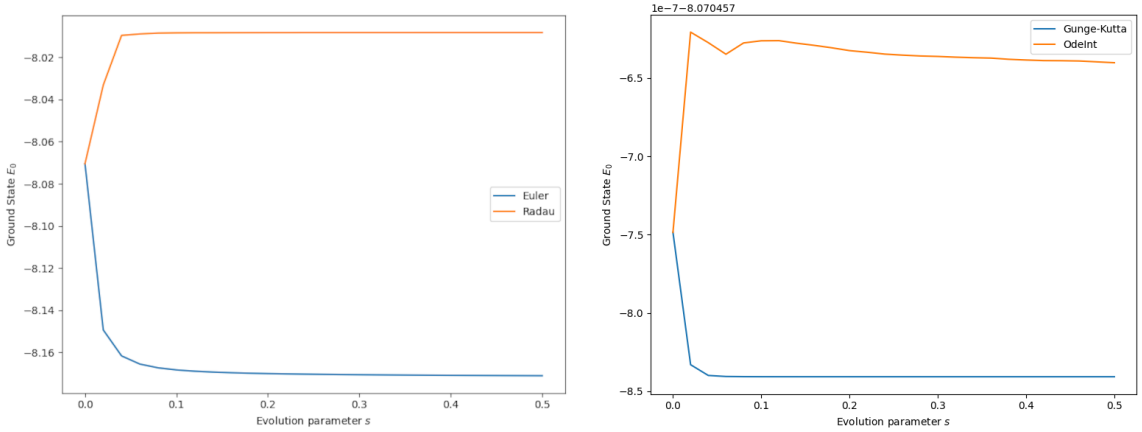

Figure 13: Comparison of evolved ground states between the Euler and the Radau method (on the left), and GK4 with the OdeInt routine on the right.

the manual implementation of two methods: the first-order Euler method and the fourth-order Runge-Kutta method (GK). As expected, there was a significant difference in accuracy, with the GK algorithm being much more precise, even at the seventh decimal position of the ground state. In contrast, the Euler method showed a significant variation already at the first decimal position, indicating its lower accuracy.

Both methods were compared using the same number of integration steps and final evolution parameter, meaning the same integration step. However, the GK method took four times longer than the Euler method to obtain the result. Nevertheless, the Euler method was not able to achieve the same level of accuracy as the GK method due to memory limitations.

Next, we compared the accuracy variation when changing the integration step for the GK algorithm and the OdeInt routine. Again, we observed a significant difference, with the manually implemented algorithm being very sensitive to the number of integration steps, as expected from an explicit method. In contrast, OdeInt was less affected by changes in the integration step due to its internal routine that manages the evolution.

Moving on to the final comparison, we found that the Euler method was comparable to the Radau method in terms of accuracy, while the GK4 method was comparable to the OdeInt method for a given final evolution parameter. However, the overall variation was in the opposite direction.
Finally, we compared the time taken by each algorithm to run the evolution. Radau was the fastest, followed by Euler, OdeInt, and GK. Taking into account both reliability and time consumption, we can conclude that the OdeInt routine is the most robust among the available algorithms.
It is important to note that the efficiency of a numerical method is closely related to the differential equation itself, and different equations may require different algorithms to achieve optimal performance.

5 Conclusion

This project report discusses the application of the Similarity Renormalization Group (SRG) evolution on the Triton, a 3-body system treated in the context of the No-Core Shell model. It also highlights the usefulness of the SRG method in other contexts, such as the Nucleon-Nucleon potential, where it can be used to soften the matrix elements and make the problem more tractable for numerical applications. The chosen generator effectively evolves the Hamiltonian towards a diagonal band, progressively suppressing the off-diagonal elements. Although the matrices studied in this report are limited in size, the SRG method can be crucial in simplifying large and complex matrices in real-world applications to obtain physical results.

References

[1] M. Hjorth-Jensen, M. P. Lombardo, and U. van Kolck, eds., *An Advanced Course in Computational Nuclear Physics*, vol. 936. Springer, 2017.

[2] E. Gebrerufael, *In-Medium No-Core Shell Model for Ab Initio Nuclear Structure Calculations*. PhD thesis, 2017.