Spectral Analysis of Lattice QCD Two-Point Correlation Functions

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science with Honors in
Physics from the College of William and Mary in Virginia,

by

Elena Amparo

Accepted for

HONORS
(Honors or no-Honors)

J. J. Dudek
Jozef Dudek, Advisor

Irina Novikova, Physics

Andreas Stathopoulos, Computer Science

Williamsburg, Virginia
April 26, 2018

Contents

Acknowledgments **iii**

List of Figures **iv**

List of Tables **v**

Abstract **v**

1 Introduction **1**

2 Background **2**

2.1 Correlation Matrices 2

2.2 The Generalized Eigenvalue Problem 3

2.2.1 Derivation 3

2.2.2 Properties 4

2.2.3 Spectral Decomposition of Correlation Functions 5

2.2.4 Solving The Generalized Eigenvalue Problem 7

3 Finding the Generalized Eigenvalues **9**

3.1 Statistical Methods 9

3.1.1 Jackknife Resampling 9

3.1.2 Data 10

3.2 Solving the Generalized Eigenvalue Problem for a Statistical Ensemble 11
3.2.1 Matching Eigenvalues Across Configurations 11
3.2.2 Calculating $C(t_0)$ 13
3.2.3 Solving the Generalized Eigenvalue Problem 14

4 Fitting the Eigenvalues **16**
4.1 Fit Fuction 16
4.2 Determining Goodness of Fit 16
4.2.1 Chi-squared Test 17
4.2.2 Chi-squared Probability Function 17
4.3 Fitting Procedure 18
4.3.1 Minuit 18
4.3.2 Trial Fits 18
4.3.3 Choosing t_0 19

5 Results **20**

6 Conclusion and Further Work **20**

A Code **26**
A.1 Eigenvector Reordering 26
A.2 Solving the Generalized Eigenvalue Problem 29
A.3 Fitting the Eigenvalues 35
A.4 Utility Functions 41

Acknowledgments

I would like to thank my advisor, Professor Jozef Dudek, for patiently and effectively explaining all of this material to me. I would also like to thank the rest of my examining committee, Professors Irina Novikova and Andreas Stathopoulos, for providing their time and feedback.

List of Figures

3.1 Jackknife Resampling of $C_{5,5}(t)$ 11

3.2 λ_2 scatterplot at $t_0 = 8, 12$ 15

4.1 Goodness of fit vs. t_0 23

4.2 E_n vs. t_0 24

5.1 $\lambda_n(t)$ fits 25

List of Tables

4.1 Minuit arguments. 19

5.1 Fit Parameter Values 21

5.2 Discrete Energies 21

Abstract

We extracted a discrete energy spectrum corresponding to scattering of two pions in a finite volume in Quantum Chromodynamics by analyzing matrices of two-point correlation functions computed within lattice QCD. We solved the generalized eigenvalue problem $C(t)v = \lambda C(t_0)v$, where the eigenvectors correspond to the optimal linear combination of basis operators to interpolate each state in the spectrum and the time dependence of the eigenvalues is controlled by the state energy. In order to solve the generalized eigenvalue problem, we used the properties of positive definite matrices to decompose $C(t_0)$ into its eigensystem and rewrite the GEVP as an ordinary eigenvalue problem with a Hermitian matrix. In order to fit the time dependence of the eigenvalues, it was necessary to identify corresponding eigenvalues across times and across configurations. This was accomplished by comparing the inner products of the corresponding eigenvectors.

The energies were extracted from the eigenvalues by applying nonlinear fitting of the form $\lambda(t) = (1 - A)e^{-E(t-t_0)} + Ae^{-E'(t-t_0)}$ to the to the time-dependent generalized eigenvalues of the correlation matrix, where the second exponential with the E' parameter accounts for excited states not captured by the limited basis of operators used.

Chapter 1

Introduction

Quantum Chromodynamics is a quantum field theory describing the interactions of quarks and gluons, the fundamental particles which are confined within strongly interacting composite particles called hadrons. In lattice QCD, quark and gluon fields are discretized on a space-time grid of finite volume with finite spacing. The path integral in quantum field theory is over all possible field configurations φ with weights $e^{-iS[\varphi]}$, where the action S is the integral of the Lagrangian of QCD. After transforming to Euclidean time, $it \to t$, $e^{-iS[\varphi]}$ becomes real and can be interpreted as a probability. The expectation value of an observable can be computed in lattice QCD by averaging over an ensemble of possible field configurations drawn randomly according to this probability using importance sampled Monte Carlo [1].

The discrete spectrum of interacting hadrons can be extracted from the two-point correlation functions of a basis of interpolating operators constructed from quark and gluon fields with the quantum numbers of the hadrons. A formalism was developed by Lüscher to determine the infinite-volume scattering amplitudes from the discrete spectrum of QCD in a finite cube. This formalism relates the discrete spectrum to discrete scattering phase shifts [1].

Unstable hadrons are observed experimentally as resonances, peaks in the scattering cross-sections of stable hadrons [1]. Here, we analyze correlation functions from

[2] corresponding to the ρ resonance in $\pi\pi$ scattering.

Chapter 2

Background

2.1 Correlation Matrices

A two-point correlation function $C_{ij}(t) = \langle 0|\mathcal{O}_i(t)\mathcal{O}_j(0)|0\rangle$ gives the vacuum expectation value of the product of the operators $\mathcal{O}_i(t)\mathcal{O}_j(0)$ where $\mathcal{O}_i(t)$ evolves in time while $\mathcal{O}_j(0)$ is fixed at its value at $t = 0$. The time evolution of the operator $\mathcal{O}(t)$ in the Heisenberg picture is given by $\mathcal{O}(t) = e^{i\hat{H}t}\mathcal{O}(0)e^{-i\hat{H}t}$ [3]. After transforming to Euclidean time, $it \to t$, $\mathcal{O}(t)$ becomes $\mathcal{O}(t) = e^{\hat{H}t}\mathcal{O}(0)e^{-\hat{H}t}$. By inserting a complete set of eigenstates $|n\rangle$ such that $\sum_n |n\rangle\langle n| = 1$ and $\hat{H}|n\rangle = E_n|n\rangle$, a correlation function can be decomposed as

$$C_{ij}(t) = \sum_n \langle 0|\mathcal{O}_i(t)|n\rangle\langle n|\mathcal{O}_j(0)|0\rangle = \sum_n \langle 0|e^{\hat{H}t}\mathcal{O}_i(0)e^{-\hat{H}t}|n\rangle\langle n|\mathcal{O}_j(0)|0\rangle$$

$$= \sum_n \langle 0|\mathcal{O}_i(0)|n\rangle\langle n|\mathcal{O}_j(0)|0\rangle e^{-E_n t} \tag{2.1}$$

where the overlap $\langle n|\mathcal{O}_j(0)|0\rangle$ indicates how well the operator $\mathcal{O}_j$ interpolates the eigenstate $|n\rangle$ from the vacuum. The term $e^{E_0 t}$ is a constant scale factor and is eliminated by setting $E_0 = 0$.

A correlation function is a sum of exponentials, but a multi-exponential fit is likely to fail because the curves of closely spaced energies cannot be distinguished. Instead, we will apply a variational method to extract the energies individually.

The path integral in quantum field theory is over all possible field configurations, so calculated values of $C_{ij}(t)$ are statistical ensembles $\left\{C_{ij}^k(t)\right\}_{k=1}^N$ corresponding to a sampling of possible field configurations where the mean over the ensemble provides the estimate of the observable and the variance on the mean gives a measure of the precision of the estimate.

2.2 The Generalized Eigenvalue Problem

Given a matrix of correlation functions, we would like to extract the energies of the eigenstates. Using a variational approach to find the linear combinations of operators which best approximate the eigenstates leads to a generalized eigenvalue problem.

2.2.1 Derivation

Consider a diagonal correlator $\langle 0|\Omega(t)\Omega^\dagger(0)|0\rangle$, where $\Omega = \sum_i v_i^* \mathcal{O}_i$ is a linear combination of operators. Then it can be decomposed as

$$\begin{aligned}\langle 0|\Omega(t)\Omega^\dagger(0)|0\rangle &= \sum_n \langle 0|e^{\hat{H}t}\Omega(0)e^{-\hat{H}t}|n\rangle\langle n|\Omega^\dagger(0)|0\rangle \\ &= \sum_n \langle 0|\Omega(0)|n\rangle\langle n|\Omega^\dagger(0)|0\rangle e^{-E_n t} = \sum_n |\langle n|\Omega^\dagger(0)|0\rangle|^2 e^{-E_n t} \end{aligned} \tag{2.2}$$

where $W_n = |\langle n|\Omega^\dagger(0)|0\rangle|^2 \geq 0$ for all n. The local minima occur when $W_{n\neq j} = 0$ and $W_j \neq 0$ for some j, and correspond to $\langle 0|\Omega(t)\Omega^\dagger(0)|0\rangle \propto e^{-E_j t}$. Thus we can extract the energies by finding the local minima, which correspond to the optimal linear combinations Ω for interpolating the eigenstates $|j\rangle$.

We wish to minimize

$$\langle 0|\Omega(t)\Omega^\dagger(0)|0\rangle = \sum_{i,j} v_i^* \langle 0|\mathcal{O}_i(t)\mathcal{O}_j(0)|0\rangle v_j = \sum_{i,j} v_i^* C_{ij}(t) v_j$$

by varying the v_i. Imposing the normalization condition

$$\sum_{i,j} v_i^* \langle 0|\mathcal{O}_i(t_0)\mathcal{O}_j(0)|0\rangle v_j = N$$

for a chosen time slice t_0 prevents the trivial solution $v_i = 0$, and we can enforce this condition using a Lagrange multiplier in the minimization problem,

$$\begin{aligned}\Lambda(v_1, \cdots, v_n, \cdots, \lambda) &= \sum_{i,j} v_i^* C_{ij}(t) v_j - \lambda \left[\sum_{i,j} v_i^* C_{ij}(t_0) v_j - N \right] \\ &= \sum_{i,j} v_i^* \left[C_{ij}(t) - \lambda C_{ij}(t_0) \right] v_j + \lambda N \end{aligned} \tag{2.3}$$

Setting the gradient to zero we obtain

$$0 = \frac{\partial \Lambda}{\partial v_i^*} = \sum_j \left[C_{ij}(t) - \lambda C_{ij}(t_0) \right] v_j \tag{2.4}$$

for each i. Thus setting each component of the gradient to zero gives

$$C(t) v = \lambda C(t_0) v \tag{2.5}$$

where

$$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}. \tag{2.6}$$

is an eigenvector and λ is an eigenvalue of the generalized eigenvalue problem. The eigenvector v gives the optimum linear combination of interpolating operators to approximate the eigenstate of λ. For $n \times n$ matrices $C(t)$ we can obtain at most n eigenvalues corresponding to the n lowest energies.

2.2.2 Properties

Let v^n, v^m denote generalized eigenvectors with generalized eigenvalues λ_n, λ_m. Since $C(t)$ is Hermitian for all t we have

$$v^{m\dagger} C(t) v^n = \lambda_n(t) v^{m\dagger} C(t_0) v^n = (v^{n\dagger} C^\dagger(t) v^m)^* = (v^{n\dagger} C(t) v^m)^*$$

$$= (\lambda_m(t) v^{n\dagger} C(t_0) v^m)^* = \lambda_m^*(t) v^{m\dagger} C(t_0) v^n$$

so

$$[\lambda_n(t) - \lambda_m^*(t)] v^{m\dagger} C(t_0) v^n = 0. \tag{2.7}$$

Real Eigenvalues

Setting $n = m$ gives $[\lambda_m(t) - \lambda_m^*(t)]v^{m\dagger}C(t_0)v^m = [\lambda_n(t) - \lambda_m^*(t)]N = 0$, so

$\lambda_m(t) = \lambda_m^*(t)$ and the generalized eigenvalues are real.

Orthogonality

If $\lambda_m(t) \neq \lambda_n(t)$, we must have $v^{m\dagger}C(t_0)v^n = 0$. Thus, without degeneracy,

$$v^{m\dagger}C(t_0)v^n = N\delta_{nm}. \tag{2.8}$$

N is arbitrary, so we may choose $N = 1$, and the eigenvectors are orthogonal on the metric $C(t_0)$. This allows us to distinguish between eigenvalues that are nearly degenerate.

From $n \times n$ matrices $C(t)$ we obtain n eigenvectors, which are forced by the GEVP to be orthogonal on the metric $C(t_0)$. This is only a good approximation to the true orthogonality if $C(t_0)$ is saturated by the n lightest states at time t_0 [4]. Thus t_0 should be chosen large enough that the contributions of higher states have decayed away.

Completeness

We have

$$\sum_m v^m v^{m\dagger}C(t_0)v^n = \sum_m v^m \delta_{mn} = v^n$$

so

$$\sum_m v^m v^{m\dagger} = C^{-1}(t_0). \tag{2.9}$$

2.2.3 Spectral Decomposition of Correlation Functions

The spectral decomposition of a two-point correlation function is given by

$$C_{ij}(t) = \langle 0|\mathcal{O}_i(t)\mathcal{O}_j(0)|0\rangle = \sum_p \frac{Z_i^{p*}Z_j^p}{2E_p}e^{-E_p t} \tag{2.10}$$

where $Z_i^p = \langle 0|\mathcal{O}_i|p\rangle$ gives the overlap factor of the eigenstate $|p\rangle$ and the interpolating operator $\mathcal{O}_i$ [4]. Let

$$v^m = \begin{bmatrix} v_1^m \\ \vdots \\ v_n^m \end{bmatrix}. \tag{2.11}$$

denote the generalized eigenvector corresponding to the eigenvalue λ_m. Then combining the spectral decomposition with the GEVP we obtain

$$C_{ij}(t)v_j^m = \sum_p \frac{1}{2E_p} Z_i^{p*} Z_j^p v_j^m e^{-E_p t} = \lambda_m(t) C_{ij}(t_0) v_j^m = \sum_p \frac{1}{2E_p} Z_i^{p*} Z_j^p v_j^m \lambda_m(t) e^{-E_p t_0}$$

for all $j \leq n$ and thus

$$\sum_p \frac{Z_i^{p*}}{2E_p} Z^p \cdot v^m \left[e^{-E_p t} - \lambda_m(t) e^{-E_p t_0}\right] = 0 \tag{2.12}$$

$$= \frac{Z_i^{m*}}{2E_m} Z^m \cdot v^m \left[e^{-E_m t} - \lambda_m(t) e^{-E_m t_0}\right] + \sum_{p \neq m} \frac{Z_i^{p*}}{2E_p} Z^p \cdot v^m \left[e^{-E_p t} - \lambda_m(t) e^{-E_p t_0}\right] \tag{2.13}$$

for all m, i. This implies that the time dependence of the eigenvalues is given by

$$\lambda_m(t) = e^{-E_m(t - t_0)} \tag{2.14}$$

for all m and the Z^p and v^m obey the orthogonality relation

$$Z^p \cdot v^m = 0 \tag{2.15}$$

for all $p \neq m$. Using the orthogonality on the metric $C(t_0)$ we obtain

$$1 = \sum_p v^{m*} Z^{p*} Z^p v^m \frac{e^{-E_p t_0}}{2E_p} = \frac{1}{2E_m} |Z^m v^m|^2 e^{-E_m t_0} \tag{2.16}$$

and thus

$$\sum_i Z_i^m v_i^p = \sqrt{2E_m} e^{\frac{E_m t_0}{2}} \delta_{mp} \tag{2.17}$$

Then we can use the completeness relation to solve for the overlap factors. We have

$$\sum_i Z_i^m \sum_p v_i^p v_j^{p*} = \sqrt{2E_m} e^{\frac{E_m t_0}{2}} v_j^{m*} \tag{2.18}$$

and substituting the completeness relation we obtain

$$Z_i^m = \sqrt{2E_m} e^{\frac{E_m t_0}{2}} \left[v^{m\dagger} C(t_0) \right]_i . \tag{2.19}$$

In principle this allows us to find the energies from a single exponential fit of the eigenvalues and to reconstruct the correlation functions from the energies and eigenvalues. However, because we can only calculate finitely many eigenvalues, the eigenvalues that we calculate contain contributions from the larger energies at small t before they have decayed away. To account for this, we will need to fit the eigenvalues with a sum of two exponentials,

$$\lambda_n^{fit}(t) = (1 - A_n) e^{-E_n(t-t_0)} + A_n e^{-E_n'(t-t_0)}, \tag{2.20}$$

where $E_n' > E_n$ and E_n determines $\lambda_n(t)$ in the limit as $t \to \infty$ and the higher energies decay away.

Equation (2.19) only allows us to reconstruct $C_{ij}(t)$ completely in the case where the $\dim(C)$ lightest states dominate $C_{ij}(t_0)$ [4]. At large t, the lowest energies dominate because they decay more slowly; however, statistical fluctuations also increase at large t. The parameter t_0 must be strategically chosen to balance these considerations.

2.2.4 Solving The Generalized Eigenvalue Problem

In order to extract the energy spectrum, we must solve the generalized eigenvalue problem $C(t)v = \lambda C(t_0)v$. (For this discussion, assume that t and t_0 are fixed). Naively, this can be transformed into an ordinary eigenvalue problem as $C(t_0)^{-1}C(t)v = \lambda v$. However, $C(t_0)^{-1}C(t)$ is not necessarily Hermitian. We would prefer to solve

the eigenvalue problem for a Hermitian matrix because it is computationally easier and the resulting eigenvectors will be mutually orthogonal. We can transform the GEVP into an ordinary eigenvalue problem with a Hermitian matrix by exploiting the positive-definiteness of $C(t_0)$.

A symmetric positive-definite matrix has real positive eigenvalues and a complete set of orthogonal eigenvectors, so it can be decomposed as $C(t_0) = U\Sigma U^T$ where the columns of U are the normalized eigenvectors and Σ is a diagonal matrix with the eigenvalues along the diagonal. The eigenvectors are orthonormal, so they satisfy $U^T U = UU^T = I$. Then we can transform the GEVP to

$$\left(\frac{1}{\sqrt{\Sigma}} U^T C(t) U \frac{1}{\sqrt{\Sigma}}\right) \sqrt{\Sigma} U^T v = \lambda \sqrt{\Sigma} U^T v \tag{2.21}$$

Now the GEVP has been reduced to the ordinary eigenvalue problem $\tilde{C}(t)\tilde{v} = \lambda\tilde{v}$, where $\tilde{C}(t) = \frac{1}{\sqrt{\Sigma}} U^T C(t) U \frac{1}{\sqrt{\Sigma}}$ is Hermitian and $\tilde{v} = \sqrt{\Sigma} U^T v$. The generalized eigenvectors can be recovered as $v = U\frac{1}{\sqrt{\Sigma}}\tilde{v}$.

Non-Positive-Definiteness

$C(t_0)$ should be positive definite, but statistical fluctuations due to the finite size of the ensemble can lead small positive eigenvalues to fluctuate into nonpositive values. If there are any nonpositive eigenvalues, $\sqrt{\Sigma}$ and $\frac{1}{\sqrt{\Sigma}}$ become complex or undefined, and $\tilde{C}(t)$ is not real. These cases must be dealt with in a systematic manner. One strategy is to eliminate the $C(t_0)$ eigenvectors which correspond to the nonpositive eigenvalues and solve the resulting lower-dimensional eigenvalue problem.

Suppose $C(t_0)$ is an $n \times n$ matrix with m positive eigenvalues. Let V denote the $n \times m$ submatrix of U corresponding to eigenvectors with positive eigenvalues, and let Υ denote the $m \times m$ submatrix of Σ corresponding to the positive eigenvalues. Let $\tilde{C}(t)$ be redefined as $\tilde{C}(t) = \frac{1}{\sqrt{\Upsilon}} V^T C(t) V \frac{1}{\sqrt{\Upsilon}}$. This is consistent with the earlier

definition when $C(t_0)$ is positive definite. Now $\tilde{C}(t)$ is a real Hermitian matrix, as desired. The size of $\tilde{C}(t)$ is now $m \times m$, so we will only be able to calculate the m lowest energies corresponding to the m largest eigenvalues.

Chapter 3

Finding the Generalized Eigenvalues

3.1 Statistical Methods

3.1.1 Jackknife Resampling

In order to correctly propagate the statistical errors due to the finite ensemble size, we used a single-elimination jackknife. For a value x_i in an ensemble of size N with mean $\bar{x}$, the "jackknife-scaled down" value $\check{x}_i$ is the mean of the ensemble with x_i removed:

$$\check{x}_i = \frac{1}{N-1}\sum_{j\neq i} x_j = \frac{1}{N-1}\left(\sum_j x_j - x_i\right) = \frac{1}{N-1}\left(N\bar{x} - x_i\right) = \bar{x} - \frac{x_i - \bar{x}}{N-1}. \quad (3.1)$$

The jackknife-scaled down ensemble has the same mean as the original ensemble but a smaller variance because the scaling-down operation brings all of the values close to the mean.

The inverse operation gives the jackknife-scaled up value, $\hat{x}_i = \bar{x} - (N-1)(x_i - \bar{x})$. The jackknife-scaled up ensemble has the same mean as the original ensemble but a larger variance. Applying scaling-up to the scaled-down ensemble gives the original ensemble.

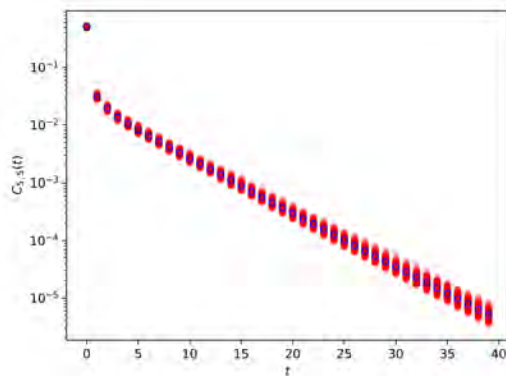
Let $F(p_1, ..., p_n)$ be a quantity calculated from n parameters, where the set of parameters $(p_1, ..., p_n)$ corresponds to an ensemble of N configurations $\{(p_{1_k}, ..., p_{n_k})\}_{k=1}^{N}$. To obtain the jackknife estimate of F, F is calculated N times using the scaled-down parameters $\{(\check{p}_{1_k}, \cdots, \check{p}_{n_k})\}_{k=1}^{N}$. The N estimates of F are then jackknife-scaled up. The mean and variance of $\left\{\hat{F}(\check{p}_{1_k}, \cdots, \check{p}_{n_k})\right\}_{k=1}^{N}$ are the jackknife estimates of the mean and variance obtained by calculating F using the unscaled parameters and propagating the errors analytically. Jackknife resampling is a robust method for calculating the variance of a Monte Carlo sample average [1].

3.1.2 Data

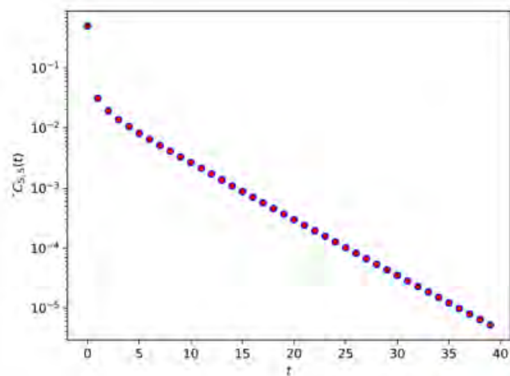
Our analysis data are a set of previously calculated two-point correlation functions, originally analyzed in [2]. The data consist of 469 configurations of the two-point correlation functions $C_{ij}(t)$ of 16 operators as they evolve over 40 discrete time slices. Fig. 3.1 (a) shows the ensemble data corresponding to $C_{5,5}(t)$.

Before the data were analyzed, each data point was jackknife-scaled down. Fig. 3.1 (b) shows the jackknife-scaled down ensemble values corresponding to $C_{5,5}(t)$. As the data show, jackknife-scaling down does not change the mean but reduces the variance.

After scaling down, the matrices $\left\{\left[C^k(t)\right]_{t=0}^{39}\right\}_{k=1}^{469}$ were symmetrized in i, j by averaging the upper- and lower-triangular portions. With a finite ensemble, the symmetric nature of $C(t)$ becomes subject to statistical fluctuations. Our calculations rely on the symmetric nature of $C(t)$, so it is necessary to enforce this condition.



(a) $C_{5,5}(t)$ Raw values



(b) $C_{5,5}(t)$ Jackknife-scaled down values

Figure 3.1: The ensemble values of $C_{5,5}(t)$ are plotted in red, and the averaged values are plotted in blue. The scaled-down ensemble shows a smaller variance.

3.2 Solving the Generalized Eigenvalue Problem for a Statistical Ensemble

3.2.1 Matching Eigenvalues Across Configurations

Our $C(t_0)$ is a statistical ensemble, and thus its eigenvalues and eigenvectors are also statistical ensembles. The `numpy.linalg.eigh()` function returns eigenvalues and their corresponding eigenvectors in ascending order, but statistical fluctuations can cause eigenvalues to move to different positions in the ordering. Thus, it is necessary to check whether the order of the eigenvectors is consistent across time slices and across configurations and to reorder them if necessary.

Reordering Algorithm

Given an ensemble of N sets of n eigenvectors $\{V_k = (v_{1_k}, ..., v_{n_k})\}_{k=1}^N$ from different time slices or configurations and a reference set of eigenvectors V_r, we used the following procedure to match the ordering of each set of eigenvectors in the ensemble to the reference. For each V_k in the ensemble, we took the product $V_r^T V_k$. Then $(V_r^T V_k)_{ij} = v_{i_r}^T v_{j_k}$. The matrices V_r and V_k should have the same orthonormal

columns, possibly antiparallel or in different orders, so the absolute value of $V_r^T V_k$ should be some permutation of the columns of the identity matrix. However, due to statistical fluctuations, eigenvectors from different configurations are not exactly orthonormal, and eigenvectors from the same configuration do not remain exactly orthonormal as t increases. As a result, the entries of $V_r^T V_k$ also fluctuate.

Given the absolute value matrix $\mathrm{Abs}(V_r^T V_k)$ we used the following algorithm to produce a template for reordering the eigenvectors of V_k. We initialized an $n \times n$ matrix filled with zeroes to hold the reordering information. We located the largest element in $\mathrm{Abs}(V_r^T V_k)$ using the `numpy.argmax()` function, and replaced the corresponding entry in the template matrix with a one. We then replaced the entries in the corresponding row and column of $\mathrm{Abs}(V_r^T V_k)$ with zeroes. We repeated this process until the matrix initialized as $\mathrm{Abs}(V_r^T V_k)$ contained only zeroes. The final template matrix contained at most one one in each row or column.

If the n-th column of the template had a one in the j-th position, the n-th eigenvector was moved to the j-th position. If the n-th column and the n-th row of the template contained all zeroes, the n-th eigenvector remained in the n-th position the new ordering. If the n-th column contained all zeroes but the n-th row contained a one in the j-th position, we followed the cycle $(j_1 j_2 \cdots j_m)$ where $j_1 = j$, j_{k+1} is given by the position of the one in row j_k, and j_m indicates the all-zero row that terminates the cycle. The n-th eigenvector was then moved to the j_m-th position.

Following a suggestion in [4], we used the eigenvectors from the average configuration as the reference. A more detailed explanation is given in sections 3.2.2 and 3.2.3, and the corresponding code is in Sec. A.1.

3.2.2 Calculating $C(t_0)$

After jackknife rescaling and symmetrizing the correlation matrices, we used the `numpy.linalg.eigh()` function to find the eigenvalues and eigenvectors of each configuration of $C(t_0)$. We also calculated the eigenvalues and eigenvectors of the average $C^a(t_0)$ of the scaled-down and symmetrized configurations. We reversed the order of the eigenvalues and eigenvectors so that they would be arranged in descending order. We then applied our reordering algorithm to the ensemble using the average eigenvectors as the reference.

Positive-Definiteness

Although $C(t_0)$ should be positive definite, statistical fluctuations can cause small positive eigenvalues to fluctuate into nonpositive values. We discussed how to transform the resulting non-positive definite matrix into a smaller positive definite matrix in Sec. 2.2.4.

Using the `test_non_pos_def()` function in Sec. A.2, we can identify the time slices t_0 that result in non-positive definite configurations. The function returns the following output,

```
Nonpositive eigenvalue of -27.291601351133917 at t0=0, index=13, N_configurations=469
Nonpositive eigenvalue of -5.130144861983919e-05 at t0=17, index=15, N_configurations=374
Nonpositive eigenvalue of -0.00018879900198779753 at t0=18, index=14, N_configurations=460
Nonpositive eigenvalue of -0.00020619884341343726 at t0=19, index=14, N_configurations=469
Nonpositive eigenvalue of -0.0005175618889847198 at t0=20, index=14, N_configurations=469
Nonpositive eigenvalue of -1.961218051902214e-06 at t0=21, index=11, N_configurations=1
Nonpositive eigenvalue of -0.00034110957418619157 at t0=22, index=13, N_configurations=469
Nonpositive eigenvalue of -0.00021119047751290535 at t0=23, index=13, N_configurations=469
Nonpositive eigenvalue of -0.0002833072607187544 at t0=24, index=13, N_configurations=469
Nonpositive eigenvalue of -8.457823352329674e-05 at t0=25, index=13, N_configurations=468
Nonpositive eigenvalue of -0.00015326501155486037 at t0=26, index=13, N_configurations=469
Nonpositive eigenvalue of -8.074357783843352e-05 at t0=27, index=12, N_configurations=469
Nonpositive eigenvalue of -6.0892296480923185e-05 at t0=28, index=12, N_configurations=468
Nonpositive eigenvalue of -6.663007751460674e-05 at t0=29, index=12, N_configurations=20
Nonpositive eigenvalue of -0.00017839444776630273 at t0=30, index=12, N_configurations=469
Nonpositive eigenvalue of -6.582545416230192e-05 at t0=31, index=12, N_configurations=75
Nonpositive eigenvalue of -8.844588992290538e-06 at t0=32, index=10, N_configurations=4
Nonpositive eigenvalue of -0.0002535737386193676 at t0=33, index=13, N_configurations=469
```

```
Nonpositive eigenvalue of -0.00012086115360956511 at t0=34, index=12, N_configurations=469
Nonpositive eigenvalue of -7.496014033210394e-06 at t0=35, index=12, N_configurations=29
Nonpositive eigenvalue of -0.0002499499134225854 at t0=36, index=12, N_configurations=469
Nonpositive eigenvalue of -7.259141546309422e-05 at t0=37, index=11, N_configurations=469
Nonpositive eigenvalue of -1.3671939252197858e-05 at t0=38, index=11, N_configurations=281
Nonpositive eigenvalue of -0.0004985517182481691 at t0=39, index=12, N_configurations=469
```

where `index` gives the index of the largest nonpositive eigenvalue when the eigenvalues are ordered from largest to smallest, and `N_configurations` gives the number of configurations with a nonpositive eigenvalue at the specified index. The output shows that for $t_0 = 0$ and $t_0 \geq 17$, there are nonpositive eigenvalues, usually in a large number of configurations. The index of the largest nonpositive eigenvalue also decreases with t_0, reflecting the increased number of nonpositive eigenvalues due to statistical noise at large t. Although we could correct these configurations with the method described in Sec. 2.2.4, these results suggest that we should avoid these values of t_0 altogether and fit in the range of smaller t_0 where there are fewer statistical fluctuations.

3.2.3 Solving the Generalized Eigenvalue Problem

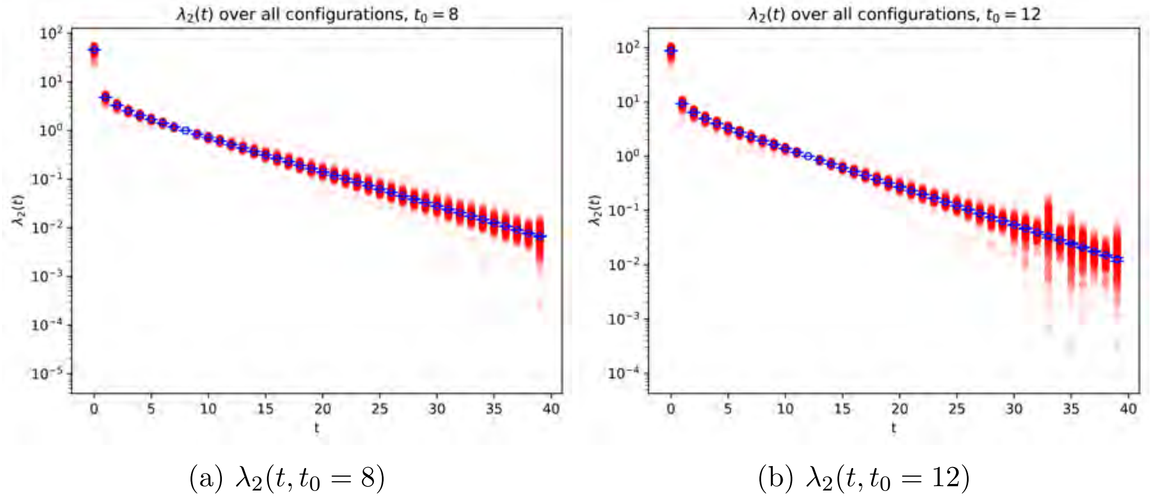
For each time slice, we constructed the ensemble $\left\{\tilde{C}^k(t)\right\}_{k=1}^{469}$ using the positive-definite portion of the eigensystems of $\left\{C^k(t_0)\right\}_{k=1}^{469}$ as described in sections 2.2.4 and 3.2.2. For each time slice, we also calculated the average $\tilde{C}^a(t)$ using the average of the scaled-down and symmetrized configurations of $C(t)$ and the positive-definite portion of the eigensystem of the average $C^a(t_0)$ calculated in section 3.2.2. We then used `numpy.linalg.eigh()` to calculate the eigenvalues and eigenvectors of $\left\{\tilde{C}^k(t)\right\}_{k=1}^{469}$ and $\tilde{C}^a(t)$ for each t independently. We reversed the eigenvector order for timeslices greater than t_0 so that they were arranged in descending order. We applied our reordering algorithm to the set of averages $\left\{\tilde{C}^a(t)\right\}_{t=0}^{39}$ using $\tilde{C}^a(t_0+1)$ as the reference. For each t, we then applied our reordering algorithm to $\left\{\tilde{C}^k(t)\right\}_{k=1}^{469}$

using $\tilde{C}^a(t)$ as the reference.

Our calculations produced an ensemble of values of $\{\lambda_n^k(t)\}_{k=1}^{469}$ for each eigenvalue λ_n and each timeslice t. We calculated the average value for each $\lambda_n(t)$ and jackknife-scaled up each ensemble value. For each λ, we then used the scaled-up values to calculate the covariance matrix given by

$$\mathbb{C}_{i,j} = \sum_{k=1}^{469} \frac{(\lambda^k(t_i) - \bar{\lambda}(t_i))(\lambda^k(t_j) - \bar{\lambda}(t_j))}{N(N-1)}. \tag{3.2}$$

In the formula, we divide by N to calculate the variance of the mean. The scatter plots in Fig. 3.2 show the scaled-up ensemble eigenvalues of λ_2 for $t_0 = 8$ and $t_0 = 12$, which will be fit in the next section. The steep upwards curve at small t which flattens into a straight line at large t corresponds to the higher energies which leak into the eigenvalues due to the finite number of basis operators. We will fit a sum of two exponentials to account for the contributions from higher energies.



(a) $\lambda_2(t, t_0 = 8)$

(b) $\lambda_2(t, t_0 = 12)$

Figure 3.2: The ensemble scaled-up values of $\lambda_2(t, t_0)$ for $t_0 = 8, 12$. The larger t_0 value shows a larger range of values, although the errors are very small in both cases.

Chapter 4

Fitting the Eigenvalues

4.1 Fit Fuction

The energies are related to the eigenvalues by

$$E_n(t_0) = -\lim_{t\to\infty} \log \frac{\lambda_n(t+1,t_0)}{\lambda_n(t,t_0)} \tag{4.1}$$

[5]. As $t \to \infty$, $\lambda_n(t)$ approaches a pure exponential, but at small t, higher energies leak in due to the finite eigenvector basis. To account for this, we fit the eigenvalues to a function of the form

$$\lambda_n^{fit}(t) = (1-A_n)e^{-E_n(t-t_0)} + A_n e^{-E'_n(t-t_0)}. \tag{4.2}$$

The E'_n in the second term accounts for the leaking in of higher energies. The contribution of E'_n decays faster because $E'_n > E_n$, so E_n determines the behavior in the limit as t goes to infinity. The parameter A_n and the factors of $t - t_0$ ensure that $\lambda_n(t_0) = 1$ as required by the generalized eigenvalue problem.

4.2 Determining Goodness of Fit

In order to determine the best value of t_0 at which to solve the GEVP and the best range $[t_{min}, t_{max}]$ within which to fit each eigenvalue, we compared fits at different t_0, t_{min}, t_{max} values using a chi-squared distribution.

4.2.1 Chi-squared Test

A good fit function $\lambda^{fit}(t)$ for $t_{min} \leq t \leq t_{max}$ should minimize the chi-squared test function

$$\chi^2 = \sum_{\substack{t_{min} \leq t_i, t_j \leq t_{max} \\ t_i, t_j \neq t_0}} \left[\lambda^{fit}(t_i) - \bar{\lambda}(t_i)\right] \left(\mathbb{C}^{-1}\right)_{i,j} \left[\lambda^{fit}(t_j) - \bar{\lambda}(t_j)\right]. \tag{4.3}$$

The number of degrees of freedom is the number of data points minus the number of parameters, $N_{dof} = t_{max} - t_{min} - 2$. A value of $\chi^2/N_{dof} \approx 1$ indicates that $\lambda^{fit}(t)$ is a good fit. However, when comparing fits with different numbers of degrees of freedom, it is more appropriate to use a chi-squared probability function.

4.2.2 Chi-squared Probability Function

The incomplete gamma function $Q(a, x)$ is defined

$$Q(a, x) \equiv \frac{1}{\Gamma(a)} \int_x^\infty e^{-t} t^{a-1} dt \qquad (a > 0). \tag{4.4}$$

The complementary cumulative distribution function $F(\chi^2|\nu)$, or tail distribution, of the chi-squared distribution is the probability that the observed chi-square of a model will exceed the value χ^2 by chance even if the model is correct [6].

The tail distribution function $F(\chi^2|\nu)$ is related to the incomplete gamma function by

$$F(\chi^2|\nu) = Q\left(\frac{\nu}{2}, \frac{\chi^2}{2}\right), \tag{4.5}$$

where ν is the number of degrees of freedom. We used `scipy.special.gammaincc()` to evaluate the incomplete gamma function.

The limiting values are $F(0|\nu) = 1$ and $F(\infty|\nu) = 0$. Thus, the best fit has the value of $F(\chi^2|\nu)$ closest to 1, which is the largest value [6].

4.3 Fitting Procedure

We fit the five largest eigenvalues for comparison with [2].

4.3.1 Minuit

To minimize the chi-squared test function, we used the `iminuit` package, which implements Minuit in Python. To fit a given eigenvalue for a given t_0, t_{min}, and t_{max}, we constructed the χ^2 function and created a Minuit object with the arguments given in Table 4.1. After the first successful fit of each eigenvalue at each t_0, we used the calculated values of E_n and E'_n from the previous successful fit as guesses in subsequent fits over different time ranges. If there were no existing fits for the current t_0, we used the E_n calculated at $t_0 - 1$ if it existed.

We called `migrad()` on the Minuit object to perform the fit. After `migrad()` we checked whether the function minimum was valid using `m.get_fmin().is_valid`. For invalid minima, we discarded the data. For valid minima, we called `m.hesse()` to recalculate the parabolic errors and discarded the fit data if the error calculation failed. Minuit returns the minimum value of χ^2, the parameter values at the minimum, and the parameter errors, which we denote by δ. We discarded the fit data if $\chi^2/N_{dof} > 10$, $F(\chi^2|\nu) < 10^{-10}$, $E'_n < E_n$, $\frac{\delta E_n}{E_n} > 2$, or any of the parameters errors were less than 10^{-10}, because these zero errors indicate that the fit did not properly converge.

4.3.2 Trial Fits

To find the best fit for a particular eigenvalue at a fixed t_0, we used Minuit to minimize the χ^2 function for various values of t_{min} and t_{max}. To determine the overall fitting range, we calculated the fractional errors $\frac{\sigma\lambda(t)}{\lambda(t)}$ at each t and fit within the band around t_0 where the fractional errors were less than 0.15. Within this band, we tested values

Minuit object argument	Value
E_n guess	0.1*
E_n initial step size	0.00001
E_n limits	$(0, 1)$
E'_n guess	0.6*
E'_n initial step size	0.00001
E'_n limits	$(0, 2)$
A_n guess	0.2
A_n initial step size	0.00001
A_n limits	$(0, 1)$
errordef	1
strategylevel	2

Table 4.1: Initial values and constraints on parameters in Minuit object. For a χ^2 fit, errrordef = 1 gives the correct errors. A strategy level of 2 produces the most accurate fit. *Previously calculated values of E_n and E'_n were used where possible.

of t_{min} in the range $[2, 5]$ to ensure that there were enough data points from small t values to accurately fit the higher energy, and t_{max} in the range $[(t_{min} + 25), 39]$ to ensure that a sufficiently large set of data points was used in each fit. For each fit, we recorded the minimum χ^2 reported by Minuit and calculated $F(\chi^2|N_{dof})$. After running all of the trial fits for a given t_0, we selected the undiscarded fits with the smallest χ^2/N_{dof} and separately selected the undiscarded fits with the largest $F(\chi^2|N_{dof})$.

4.3.3 Choosing t_0

For each eigenvalue to be fit, we tested t_0 in the range $2 - 12$. For each value of t_0, we used the method in section 4.3.2 to find the best fits for that t_0 based on χ^2/N_{dof} and $F(\chi^2|N_{dof})$. Figure 4.1 shows the goodness of fit as a function of t_0. Figure 4.2 show the energies E_n as a function of t_0. A missing plot point for a given t_0 indicates that all fits were discarded for the corresponding eigenvalue and t_0.

Figure 4.2 shows that most t_0 values lead to approximately the same calculation

of E_n. Figure 4.1 shows that the goodness of fit generally improves with increasing t_0, although for the lowest energy the goodness of fit falls at $t_0 = 9$. Based on these considerations, we chose $t_0 = 8$ for our reported values.

Chapter 5

Results

Figure 5.1 shows the fits for $t_0 = 8$. The fits are plotted as $e^{E_n(t-t_0)}\lambda_n(t)$. The decay at small t shows the excited-state contribution $e^{-E'_n(t-t_0)}$ decaying away, and the flattening at large t shows $\lambda_n(t)$ approaching a pure exponential.

The fit parameters are given in Table 5.1. The parameters corresponding to energies are in the dimensionless form a_tE, a_tE' because the integer timeslices used in the fitting procedure correspond to t/a_t, where a_t is the lattice spacing $a_t \approx (5997\ \mathrm{MeV})^{-1}$ [2]. Table 5.2 gives the discrete energies.

Chapter 6

Conclusion and Further Work

We have calculated the lowest five states of the discrete spectrum of $\pi\pi$ scattering using a basis of interpolating operators. Beginning with a matrix of two-point correla-

Parameter	Value
a_tE_0	$(10.69 \pm 0.01)10^{-2}$
a_tE_0'	$(65.60 \pm 3.86)10^{-2}$
A_0	$(5.77 \pm 1.24)10^{-3}$
t_{min}, t_{max}	$3, 31$
χ^2/N_{dof}	1.21
$F(\chi^2\|N_{dof})$	0.211
a_tE_1	$(14.28 \pm 0.03)10^{-2}$
a_tE_1'	$(49.41 \pm 1.72)10^{-2}$
A_1	$(27.90 \pm 2.86)10^{-3}$
t_{min}, t_{max}	$4, 39$
χ^2/N_{dof}	1.66
$F(\chi^2\|N_{dof})$	0.0099
a_tE_2	$(16.20 \pm 0.04)10^{-2}$
a_tE_2'	$(58.65 \pm 2.65)10^{-2}$
A_2	$(16.83 \pm 2.43)10^{-3}$
t_{min}, t_{max}	$4, 39$
χ^2/N_{dof}	1.42
$F(\chi^2\|N_{dof})$	0.054

Parameter	Value
a_tE_3	$(18.49 \pm 0.01)10^{-2}$
a_tE_3'	$(74.26 \pm 3.66)10^{-2}$
A_3	$(7.25 \pm 1.24)10^{-3}$
t_{min}, t_{max}	$4, 29$
χ^2/N_{dof}	1.24
$F(\chi^2\|N_{dof})$	0.198
a_tE_4	$(19.04 \pm 0.06)10^{-2}$
a_tE_4'	$(63.48 \pm 3.23)10^{-2}$
A_4	$(16.49 \pm 2.88)10^{-3}$
t_{min}, t_{max}	$4, 39$
χ^2/N_{dof}	1.27
$F(\chi^2\|N_{dof})$	0.139

Table 5.1: Fit parameter values for $\lambda_n(t)$ when $t_0 = 8$. The parameters a_tE, a_tE' are dimensionless.

E_0	641.1 ± 0.6 MeV
E_1	856.4 ± 1.8 MeV
E_2	971.5 ± 2.4 MeV
E_3	1108.8 ± 0.6 MeV
E_4	1141.8 ± 3.6 MeV

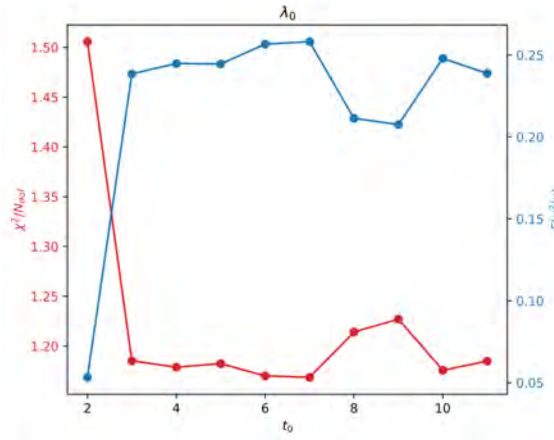
Table 5.2: The five lowest discrete energies.

tion functions, we solved the generalized eigenvalue problem to find the eigenvectors which gave the optimal linear combinations to approximate the eigenstates of the system, and we extracted the energies from the time-dependence of the eigenvalues. In [2] these energies are also found and are used to calculate scattering phase shifts corresponding to the ρ resonance.

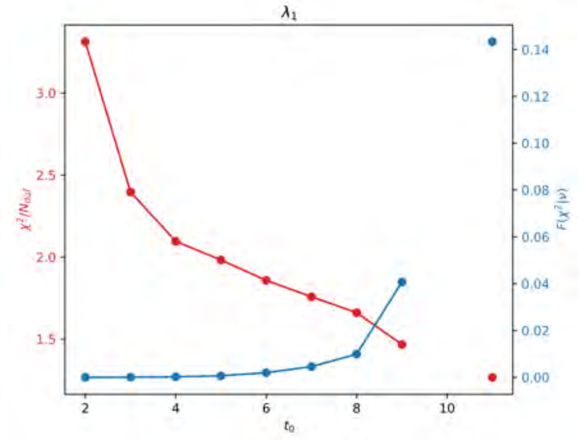
In order to solve the GEVP, it was necessary to specify a t_0, and we chose the value

of t_0 that maximized the chi-squared tail distribution across the set of eigenvalues. We can also measure directly how well the generalized eigenvectors at t_0 interpolate the eigenstates by reconstructing the correlation matrix using Eq. (2.19), as is done in [4].

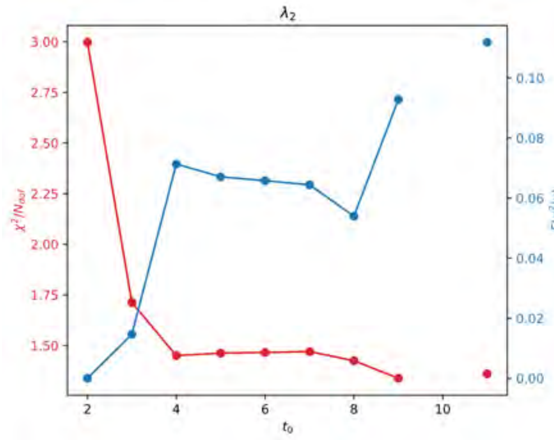
The errors on the energies are surprisingly small, approximately one thousandth of the energies. The minima could be more thoroughly investigated using contour plots and other tools in Minuit. The uncertainty in the energies could also be estimated using jackknife resampling, by fitting the scaled-down data one ensemble at a time and using the variance of the scaled-up parameters to estimate the parameter errors.



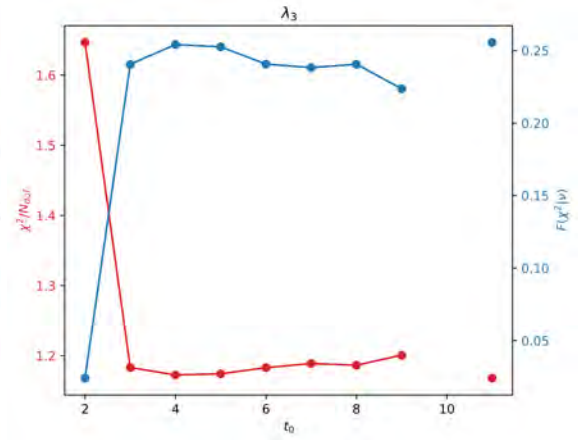
(a) λ_0



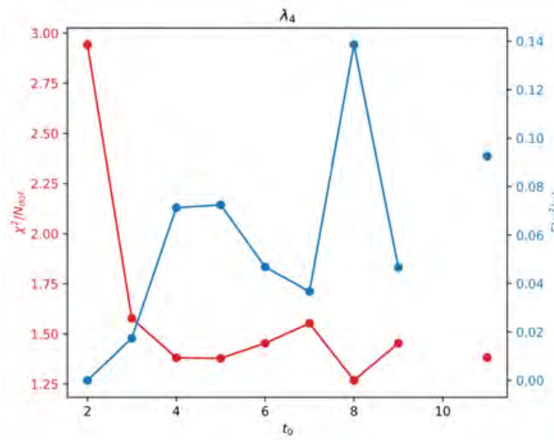
(b) λ_1



(c) λ_2

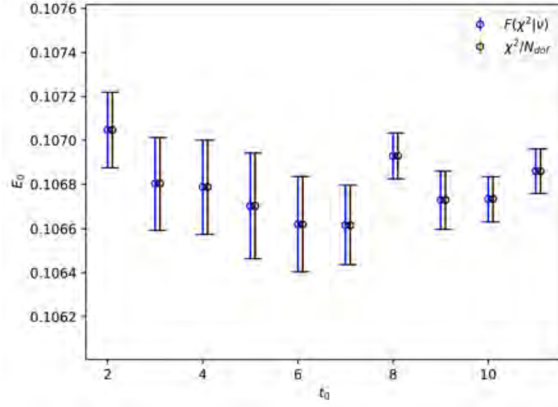


(d) λ_3

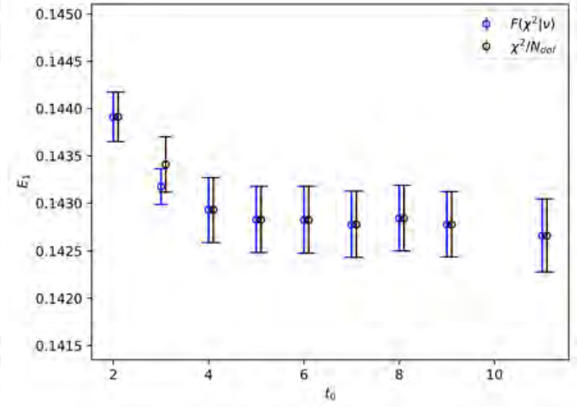


(e) λ_4

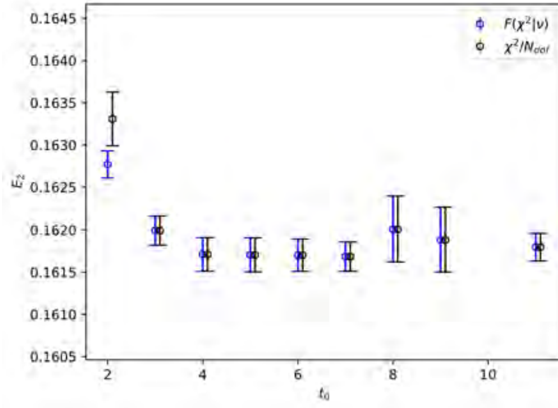
Figure 4.1: The largest value of $F(\chi^2|\nu)$ and the smallest χ^2/ν in fitting λ_n at each t_0.



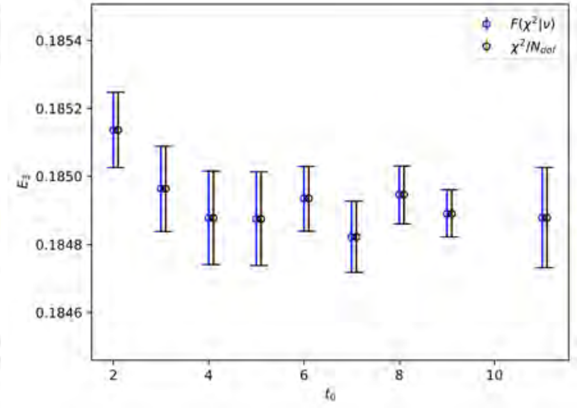
(a) E_0



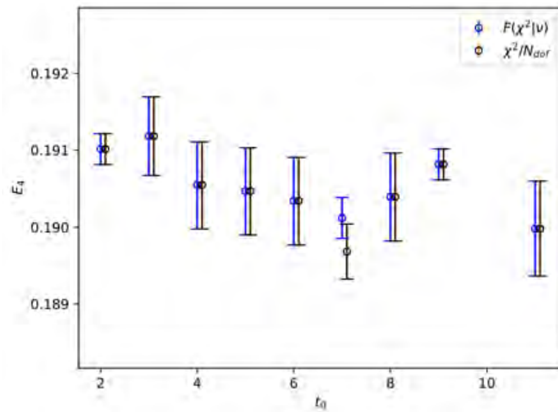
(b) E_1



(c) E_2

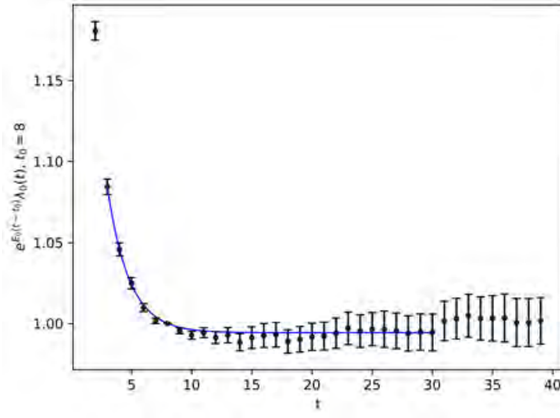


(d) E_3

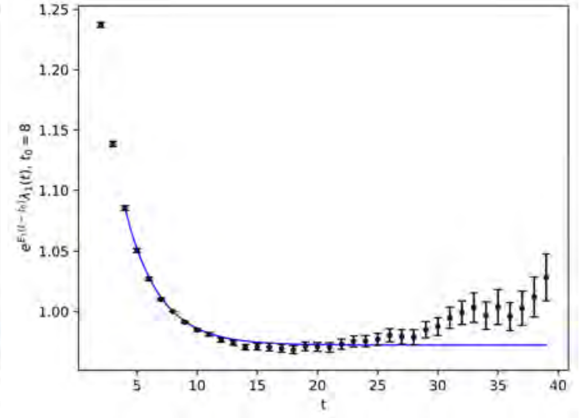


(e) E_4

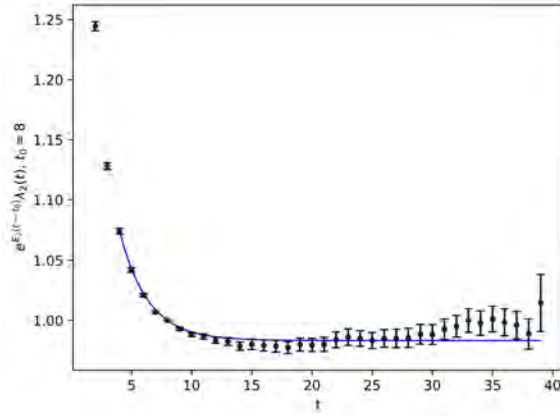
Figure 4.2: The calculated value of E_n for each t_0 with the largest $F(\chi^2|\nu)$, and the calculated value of E_n for each t_0 with the smallest χ^2/ν.



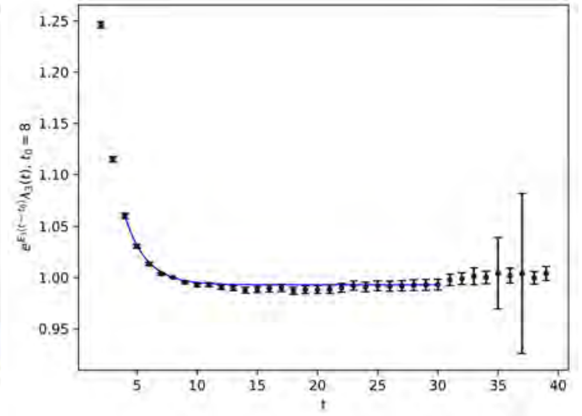
(a) λ_0



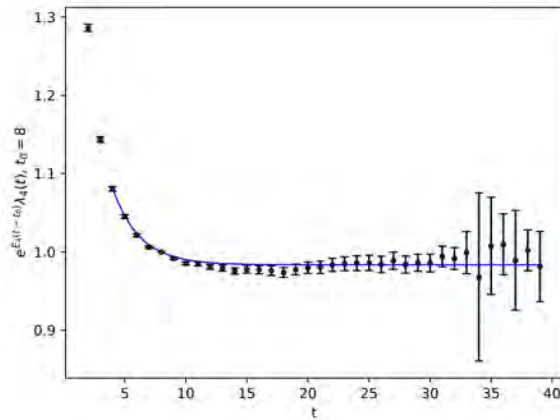
(b) λ_1



(c) λ_2



(d) λ_3



(e) λ_4

Figure 5.1: The fits with largest $F(\chi^2|\nu)$ calculated at $t_0 = 8$.

Appendix A

Code

This section contains the Python code used to perform the computations. It requires the following packages to be imported:

```
import scipy
from scipy import special
import numpy as np
from numpy import linalg
from iminuit import Minuit
import math
```

A.1 Eigenvector Reordering

The code in this section is used to match eigenvector ordering across configurations and across time slices.

```
def get_reordering_matrix(ref_eigenvectors, unordered_vectors):
    """
    Returns a template for reordering unordered_vectors to match
    ref_eigenvectors. The reordering prioritizes the inner products
    with the largest absolute value.
    """
    eigendim = ref_eigenvectors.shape[1]
    product = (np.transpose(ref_eigenvectors)).dot(unordered_vectors)
    product = np.absolute(product)
    return_matrix = np.zeros([eigendim,eigendim], dtype=float)

    max_val = np.amax(product)

    while max_val>0:

        max_ind_flat = np.argmax(product)
```

```
        row_ind = max_ind_flat//eigendim #destination column
        col_ind = max_ind_flat-row_ind*eigendim #column to be moved

        return_matrix[row_ind, col_ind] = 1.0

        #replace the row and column containing the max val with zeroes
        product[row_ind,:] = np.zeros([eigendim], dtype=float)
        product[:,col_ind] = np.zeros([eigendim], dtype=float)

        max_val = np.amax(product)

    return return_matrix
#________________________________________________________________________________

def get_cycle(template, start_ind):
    """
    Returns the permutation cycle containing start_ind. If the last column
    in the sequence is all zeroes, it is mapped to the first column in the
    sequence.
    """
    cycle = [start_ind]

    current_col = template[:,start_ind]
    while np.amax(current_col)>0:
        next_ind = np.argmax(current_col)
        if next_ind==start_ind:
            return cycle
        else:
            cycle.append(next_ind)
            current_col = template[:,next_ind]

    return cycle
#________________________________________________________________________________

def cycle_reorder(cycle, eigenvals, eigenvectors):
    """
    Reorders eigenvalues and vectors following the permutation given by cycle.
    """
    length = len(cycle)

    if length==1:
        return eigenvals, eigenvectors

    else:
        return_eigenvals = np.empty(eigenvals.shape, dtype=float)
        return_vectors = np.empty(eigenvectors.shape, dtype=float)

        for i in range(len(eigenvals)):
            if i not in cycle:
                return_eigenvals[i] = eigenvals[i]
                return_vectors[:,i] = eigenvectors[:,i]
```

```
        for i in range(length-1):
            col_ind = cycle[i]
            destination = cycle[i+1]
            return_eigenvals[destination] = eigenvals[col_ind]
            return_vectors[:,destination] = eigenvectors[:,col_ind]

        first_ind = cycle[0]
        last_ind = cycle[length-1]

        return_eigenvals[first_ind] = eigenvals[last_ind]
        return_vectors[:,first_ind] = eigenvectors[:,last_ind]

        return return_eigenvals, return_vectors
#_____________________________________________________________________________

def reorder_with_template(template, eigenvals, eigenvectors):
    """
    Reorders the eigenvalues and eigenvectors following the disjoint cycles
    in the template.
    """
    eigendim = eigenvectors.shape[1]
    I = np.identity(eigendim)

    if np.array_equal(I,template) is True:
        return eigenvals, eigenvectors

    else:

        col_max = np.empty([eigendim], dtype=float)
        for i in range(eigendim):
            template_col = template[:,i]
            if np.argmax(template_col)==i:
                col_max[i] = 0
            else:
                col_max[i] = np.amax(template_col)

        while np.amax(col_max)>0:
            start_ind = np.argmax(col_max)
            cycle = get_cycle(template, start_ind)
            for i in cycle:
                col_max[i] = 0
            eigenvals, eigenvectors = cycle_reorder(cycle, eigenvals, eigenvectors)

        return eigenvals, eigenvectors
#_____________________________________________________________

def arrange_eigensystem(eigenvals_ensemble, eigenvectors_ensemble, ref_ind=0):
    """
    Reorders an ensemble of eigenvectors along the first axis using the specified
    index along that axis as the template.
```

```
    """
    #expects an eigenvector ensemble of shape (configurations, nrows, eigendim )
    #eigenvalue ensemble of shape (configurations, eigendim)
    eigenvals_shape = eigenvals_ensemble.shape
    ensemble_size = eigenvals_shape[0]
    eigendim = eigenvals_shape[1]

    reordered_eigenvals = np.empty(eigenvals_ensemble.shape, dtype=float)
    reordered_vectors = np.empty(eigenvectors_ensemble.shape, dtype=float)

    ref = eigenvectors_ensemble[ref_ind]
    I = np.identity(eigendim, dtype=float)

    #loop over ensemble
    for i in range(ensemble_size):
        template = get_reordering_matrix(ref, eigenvectors_ensemble[i])

        if np.array_equal(template, I) is True:
            reordered_eigenvals[i], reordered_vectors[i] = eigenvals_ensemble[i], eigenvectors_ens
        else:
            reordered_eigenvals[i], reordered_vectors[i] = reorder_with_template(template, eigenva

    return reordered_eigenvals, reordered_vectors
```

A.2 Solving the Generalized Eigenvalue Problem

The code in this section is used to calculate the generalized eigenvalues and eigenvectors, and the errors in the eigenvalues.

```
def solve_Ct0(array, t0, avg_array, printing=False, testing=False):
    """
    Returns enesmble and average eigenvalues and eigenvectors of Ct0 after
    removing nonpositive eigenvalues and their eigenvectors.
    """
    #expects ensemble array with shape (times, configurations, dim(C), dim(C))
    #average array with shape (times, dim(C), dim(C))
    array_shape = array.shape
    ensemble_size = array_shape[1]
    dim = array_shape[2]

    eigenvals_ensemble = np.empty([ensemble_size, dim], dtype=float)
    eigenvectors_ensemble = np.empty([ensemble_size, dim, dim], dtype=float)

    #solve average Ct0
    avg_eigenvals, avg_vectors = eigh_dec_order(avg_array[t0])

    #solve each Ct0
```

```
    for i in range(ensemble_size):
        Ct0 = array[t0, i]
        temp_eigenvals, temp_eigenvectors = eigh_dec_order(Ct0)
        #check eigenvector ordering
        template = get_reordering_matrix(avg_vectors, temp_eigenvectors)
        eigenvals_ensemble[i], eigenvectors_ensemble[i] = \
        reorder_with_template(template, temp_eigenvals, temp_eigenvectors)

    #check for nonpositive eigenvalues
    ind_shift = 0
    for j in range(dim):
        ind = j-ind_shift
        min_eigenval = np.amin(eigenvals_ensemble[:,ind])

        if min_eigenval <= 0:

       #extract nonpositive eigenvalues from the array
            if printing is True:
                nonpositive_cond = eigenvals_ensemble[:,ind] <= 0
                nonpositive_vals = np.extract(nonpositive_cond, eigenvals_ensemble[:,ind])

   #remove the nonpositive eigenvalues and their eigenvectors
            eigenvals_ensemble = np.delete(eigenvals_ensemble,ind,1)
            eigenvectors_ensemble = np.delete(eigenvectors_ensemble,ind,2)
            avg_eigenvals = np.delete(avg_eigenvals, ind)
            avg_vectors = np.delete(avg_vectors, ind, 1)

            ind_shift += 1

      #print the eigenvalue index and number of configurations
      #with a nonpositive eigenvalue at that index
            if printing is True:
                print("Nonpositive eigenvalue of "+str(min_eigenval)+\
                " at t0="+str(t0)+", index="+str(j)+",
                N_configurations="+str(nonpositive_vals.shape[0]))
                if testing is True:
                    return

    return eigenvals_ensemble, eigenvectors_ensemble, avg_eigenvals, avg_vectors

#________________________________________________________________________________________

def test_non_pos_def():
    """
    Check the ensemble of Ct0 for nonpositive eigenvalues at each t0 value,
    and print the information.
    """
    np.set_printoptions(precision=2,linewidth=2000, suppress=False, threshold=10000)

    filepath = "C"
    raw_array = get_raw_data(filepath)
```

```
    avg_array = make_avg_matrix(raw_array)
    scaled_down_array = scale_down_and_symmetrize(raw_array, avg_array)
    symm_avg_array = make_avg_matrix(scaled_down_array)

    for t0 in range(40):
        solve_Ct0(scaled_down_array, t0, symm_avg_array, printing=True, testing=True)

#________________________________________________________________________________________

def solve_GEVP(array, t0, avg_array, reordering=True):
    """
    Returns the generalized eigenvectors and eigenvalues.
    """
    #expects ensemble array with shape (times, configurations, dim(C), dim(C))
    #average array with shape (times, dim(C), dim(C))
    array_shape = array.shape
    timeslices = array_shape[0]
    ensemble_size = array_shape[1]
    dim = array_shape[2]

    t0_eigenvals, t0_eigenvectors, avg_t0_eigenvals, avg_t0_eigenvectors = \
    solve_Ct0(array, t0, avg_array)

    eigenvals_shape = t0_eigenvals.shape
    eigendim = eigenvals_shape[1]

    GEVP_eigenvals = np.empty([ensemble_size,timeslices,eigendim],dtype=float)
    GEVP_eigenvectors = np.empty([ensemble_size,timeslices,eigendim,eigendim],\
    dtype=float)
    inv_sqt_sigma_arr = np.empty([ensemble_size,eigendim,eigendim],dtype=float)

    #Solve average GEVP
    avg_GEVP_eigenvals = np.empty([timeslices,eigendim],dtype=float)
    avg_GEVP_eigenvectors = np.empty([timeslices,eigendim,eigendim],dtype=float)
    #construct average U
    U_avg = avg_t0_eigenvectors
    UT_avg = np.transpose(U_avg)
    #construct sqt_sigma and inv_sqt_sigma
    avg_sqt_sigma = np.zeros([eigendim,eigendim], dtype=float)
    avg_inv_sqt_sigma = np.zeros([eigendim,eigendim], dtype=float)
    for i in range(eigendim):
        avg_sqt_sigma[i,i] = (avg_t0_eigenvals[i])**0.5
        avg_inv_sqt_sigma[i,i] = 1/avg_sqt_sigma[i,i]
    #sove GEVP at each timeslice
    for t in range(timeslices):
        Ct = avg_array[t]
        GEVP_matrix = avg_inv_sqt_sigma.dot(UT_avg).dot(Ct).dot(U_avg).\
        dot(avg_inv_sqt_sigma)
        #solve ordinary eigenvalue problem
        eigenvals, eigenvectors = eigh_dec_order(GEVP_matrix)
        avg_GEVP_eigenvals[t] = eigenvals
```

```
    avg_GEVP_eigenvectors[t] = eigenvectors
#reverse order of eigenvectors before t0
for t in range(0,t0):
    avg_GEVP_eigenvals[t], avg_GEVP_eigenvectors[t] =
    \reverse_eigenvector_order(avg_GEVP_eigenvals[t], avg_GEVP_eigenvectors[t])
#reorder average eigenvectors to match t0+1
avg_GEVP_eigenvals, avg_GEVP_eigenvectors = \
arrange_eigensystem(avg_GEVP_eigenvals, avg_GEVP_eigenvectors, ref_ind=t0+1)
#arrange_eigensystem matches the ordering of the ensemble to the configuration
#at index=ref_ind

#solve GEVP for ensemble
for k in range(ensemble_size):
    #construct U and UT
    U = t0_eigenvectors[k]
    UT = np.transpose(U)
    #construct sqt_sigma and inv_sqt_sigma
    sqt_sigma = np.zeros([eigendim,eigendim], dtype=float)
    inv_sqt_sigma = np.zeros([eigendim,eigendim], dtype=float)
    for i in range(eigendim):
        sqt_sigma[i,i] = (t0_eigenvals[k,i])**0.5
        inv_sqt_sigma[i,i] = 1/sqt_sigma[i,i]
    inv_sqt_sigma_arr[k] = inv_sqt_sigma

    #sove GEVP at each timeslice
    for t in range(timeslices):
        Ct = array[t,k]
        GEVP_matrix = inv_sqt_sigma.dot(UT).dot(Ct).dot(U).dot(inv_sqt_sigma)
         #solve ordinary eigenvalue problem
        eigenvals, eigenvectors = eigh_dec_order(GEVP_matrix)
        GEVP_eigenvals[k,t] = eigenvals
        GEVP_eigenvectors[k,t] = eigenvectors

#reverse order of eigenvectors before t0
for k in range(ensemble_size):
    for t in range(0,t0):
        GEVP_eigenvals[k,t], GEVP_eigenvectors[k,t] =\
        reverse_eigenvector_order(GEVP_eigenvals[k,t], GEVP_eigenvectors[k,t])

I = np.identity(eigendim, dtype=float)

if reordering==True:
    #reorder ensemble at each time t to match the average configuration
    #at time t
    for j in range(timeslices):
        ref_vectors = avg_GEVP_eigenvectors[j]
        for i in range(ensemble_size):
            temp = get_reordering_matrix(ref_vectors,GEVP_eigenvectors[i,j])
            if np.array_equal(I,temp)==False:
                GEVP_eigenvals[i,j], GEVP_eigenvectors[i,j] =\
                reorder_with_template(temp, GEVP_eigenvals[i,j], GEVP_eigenvectors[i,j])
```

```
    #recover GEVP eigenvectors
    recovered_vectors = np.empty([ensemble_size,timeslices,dim,eigendim],dtype=float)
    for t in range(timeslices):
        for k in range(ensemble_size):
            recovered_vectors[k,t] = t0_eigenvectors[k].dot(inv_sqt_sigma_arr[k]).\
            dot(GEVP_eigenvectors[k,t])

    GEVP_eigenvectors = recovered_vectors

    return GEVP_eigenvals, GEVP_eigenvectors

  #________________________________________________________________________

def get_GEVP_eigenvals(t0):
    """
    Returns the ensemble of calculated eigenvalues for the specified t0.
    """
    filepath = "C"
    raw_array = get_raw_data(filepath)
    avg_array = make_avg_matrix(raw_array)
    scaled_down_array = scale_down_and_symmetrize(raw_array, avg_array)
    symm_avg_array = make_avg_matrix(scaled_down_array)

    GEVP_eigenvals, GEVP_eigenvectors = solve_GEVP(scaled_down_array, \
    t0, symm_avg_array)

    return GEVP_eigenvals
#______________________________________

    def get_scaled_up_eigenvals(GEVP_eigenvals):
    """
    Takes in the eigenvalues returned by solve_GEVP and scales them up.
    """
    #expects an array of shape (configurations, times, dim(C), dim(C))
    shape = GEVP_eigenvals.shape
    eigendim = shape[2]
    timeslices = shape[1]
    configurations = shape[0]

    #construct avg matrix
    avg = np.empty([eigendim,timeslices],dtype=float)
    for i in range(eigendim):
        for t in range(timeslices):
            total = 0.0
            for k in range(configurations):
                total += GEVP_eigenvals[k,t,i]
            avg_val = total/configurations
            avg[i,t] = avg_val

    #scale up eigenvalues
```

```
    scaled_up_eigenvals = np.empty([configurations, timeslices, eigendim],dtype=float)
    for i in range(eigendim):
        for t in range(timeslices):
            for k in range(configurations):
                scaled_up_eigenvals[k,t,i] = jackkinfe_up(GEVP_eigenvals[k,t,i], avg[i,t], configu

    return scaled_up_eigenvals
#__________________________________________
def get_eigenval_stats(scaled_up_eigenvals, t0):
    """
    Computes and returns the average matrix, covariance matrix, inverse
    covariance matrix, and the standard error on the mean.
    """
    #expects an array of shape (configurations, times, dim(C), dim(C))
    shape = scaled_up_eigenvals.shape
    eigendim = shape[2]
    timeslices = shape[1]
    configurations = shape[0]

    #construct avg matrix
    avg = np.empty([eigendim,timeslices],dtype=float)
    for i in range(eigendim):
        for t in range(timeslices):
            total = 0.0
            for k in range(configurations):
                total += scaled_up_eigenvals[k,t,i]
            avg_val = total/configurations
            avg[i,t] = avg_val

    #construct covariance matrix
    cov = np.empty([eigendim,timeslices, timeslices],dtype=float)

    for i in range(eigendim):
        for t in range(timeslices):
            for s in range(t,timeslices):
                total = 0.0
                for k in range(configurations):
                    total += (scaled_up_eigenvals[k,t,i]-avg[i,t])*(scaled_up_eigenvals[k,s,i]-avg
                cov_term = total/(configurations*(configurations-1))
                cov[i,t,s] = cov_term
                cov[i,s,t] = cov_term

    return avg, cov, get_inv_cov(cov,t0), get_std_err(cov)
#____________________________________________

def get_inv_cov(cov, t0):
    """
    Calculates the inverse covariance matrix. It is necessary to delete
    the row and column corresponding to t0 to perform the inversion. A row
    and column of zeroes are inserted back in their place so that the indexing
    is not changed.
```

```
    """
    #cov has shape (eigendim, times, times)
    eigendim = cov.shape[0]
    timeslices = cov.shape[1]

    #remove the t0 row from the covariance matrix of each eigenvalue
    sliced_arr = np.delete(cov, t0, 1)
    #remove the t0 column from the covariance matrix of each eigenvalue
    sliced_arr = np.delete(sliced_arr, t0, 2)

    inv_cov = np.empty(sliced_arr.shape, dtype=float)

    #invert the sliced arrays
    for i in range(eigendim):
        inv_cov[i] = np.linalg.inv(sliced_arr[i])

    zero_arr = np.zeros(timeslices, dtype=float)
    zero_arr_short = np.zeros(timeslices-1, dtype=float)

    #insert a row of zeros at the t0 position in the cov matrix of each eigenval
    inv_cov = np.insert(inv_cov, t0, zero_arr_short, 1)
    #insert a column of zeros at the t0 position in the cov matrix of each eigenval
    inv_cov = np.insert(inv_cov, t0, zero_arr, 2)

    return inv_cov
#____________________________
def get_std_err(cov):
    """
    Calculate the standard error on the mean from the covariance matrix.
    """
    #expects an array of shape (eigendim, times, times)
    eigendim = cov.shape[0]
    timeslices = cov.shape[1]

    std_err = np.empty([eigendim,timeslices], dtype=float)

    for i in range(eigendim):
        for t in range(timeslices):
            std_err[i,t] = np.sqrt(cov[i,t,t])

    return std_err
```

A.3 Fitting the Eigenvalues

The code in this section was used to fit the eigenvalues.

```
def eigen_fit(A, E1, E2, t0, t):
    """
```

```
    The fit function. Used to define the chi-squared test function.
    """
    return (1-A)*np.exp(-E1*(t-t0)) + A*np.exp(-E2*(t-t0))
#___________________________________________________

def chisq_term(t1,t2, avg, inv_cov, A, E1, E2, t0):
    """
    Expects avg and inv_cov to be 2D arrays corresponding to a single
    eigenvalue. Used to calculated individual terms in the sum defining
    the chi-squared test function.
    """
    return (avg[t1]-eigen_fit(A, E1, E2, t0, t1))*inv_cov[t1,t2]*\
    (avg[t2]-eigen_fit(A, E1, E2, t0, t2))
#_________________________________________________________________________

def chisq(A,E1,E2,avg,inv_cov,t0,min_t,max_t):
    """
    Expects avg and inv_cov to be 2D arrays corresponding to a single
    eigenvalue. Calculates the chi-squared test function over the fitting
    range [min_t,max_t].
    """
    func = 0.0

    for t in range(min_t,max_t+1):
        if t != t0:
            for s in range(min_t,max_t+1):
                if s != t0:
                    func = func + chisq_term(t, s, avg, inv_cov, A, E1, E2, t0)

    return func
#__________________________________________

def Q(chisq, ndof):
    """
    Calculates the tail of the chi-square distribution function.
    """
    return special.gammaincc(ndof*0.5, chisq*0.5)
#_________________________________________

def fit_eigenvals(avg_eigenvals, inv_cov, t0, tmin, tmax, E1_guess, E2_guess):
    """
    Expects avg and inv_cov to be 2D arrays corresponding to a single
    eigenvalue. Uses Minuit Migrad to find the best fit over the range
    [tmin,tmax]. Returns a 1D array containing the fit data.
    """
    #return a row of NaNs if the fit is bad
    nan_row = np.full( (10), np.nan, dtype=float)

    #chi-square test function to be minimized with 3 free parameters
    def f_i(A,E1,E2):
        return chisq(A,E1,E2,avg_eigenvals,inv_cov,t0,tmin,tmax)
```

```
#Minuit object
#set errordef to 1 for chi-square fits
m=Minuit(f_i, E1=E1_guess, error_E1=0.00001, limit_E1=(0,1), E2=E2_guess,\
error_E2=0.00001, limit_E2=(0,2), A=0.2, error_A=0.00001, limit_A=(0,1.),\
print_level=0, errordef=1)

#strategy level 2 for more accurate fitting
m.set_strategy(2)
m.migrad();

#make sure the minimum is valid
if m.get_fmin().is_valid:

    #recalculate parabolic errors
    m.hesse()

    #if error calculation fails, return NaNs
    if m.get_fmin().hesse_failed:
        print("hesse failed")
        return nan_row

    min_chisq = m.fval
    ndof = tmax-tmin-2
    chisq_per_ndof = min_chisq/ndof

    values = m.values
    errors = m.errors

    A = values["A"]
    A_err = errors["A"]
    E1 = values["E1"]
    E1_err = errors["E1"]
    E2 = values["E2"]
    E2_err = errors["E2"]

    Q_val = Q(min_chisq, ndof)

    #if chisq too big, return NaNs
    if chisq_per_ndof > 10:
        return nan_row

    #if Q is zero, return Nans
    elif Q_val <= 10**-10:
        return nan_row

    #if E2<E1, return NaNs
    elif E2-E1 <= 0:
        return nan_row

    #if any parameter errors are zero, something must have gone wrong
```

```
        #return NaNs
        elif A_err <= 10**-10 or E1_err <= 10**-10 or E2_err <= 10**-10:
            return nan_row

        #if percent error is too large, return NaNs
        elif E1_err/E1 > 2:
            return nan_row

        #if the fit passed all the checks
        else:
            return np.array([min_chisq, ndof, chisq_per_ndof, Q_val, \
            A, A_err, E1, E1_err, E2, E2_err])

    #if the fit is not valid
    else:
        print("invalid fit")
        return nan_row

#_____________________________________________

def run_fits(avg_eigenvals, inv_cov, t0, tmin_lim, tmax_lim,\
E1_guess=0.1, E2_guess=0.6):
    """
    Expects avg and inv_cov to be 2D arrays corresponding to a single
    eigenvalue. Runs fits within different subsets of the range
    [tmin_lim, tmax_lim]. Returns an array of fit data with extra space
    filled by NaNs.
    """
    #initialize best guesses for E1, E2
    E1 = E1_guess
    E2 = E2_guess

    #100 indices-space for all the run_fits
    #12 indices-tmin, tmax, fit data
    data = np.full( (100, 12), np.nan, dtype=float)

    tstart = max(2, tmin_lim)
    tstop = min(40, tmax_lim+1)

    ind = 0
    for tmin in range(tstart,5):
        for tmax in range(tmin+25,tstop):
            data[ind,0] = tmin
            data[ind,1] = tmax
            data[ind,2:] = fit_eigenvals(avg_eigenvals, inv_cov, t0, tmin, tmax, E1, E2)
            if np.isnan(data[ind,4])==False and data[ind,4]<2:
                #if the fit didn't fail and the chisquare is not too big,
                #set the new best guess for E1 and E2
                E1 = data[ind,8]
                E2 = data[ind,10]
            ind += 1
```

```
    return data
#_____________________________________________


def fit_main(eigen_ind_start, eigen_ind_end, t0_ind_start, t0_ind_end):
    """
    Fits the eigenvalues in the range at each of the t0 bins in the range.
    Saves all successful fit data.
    """

    # 16 eigenvals, 40 times, 12 data points for each fit
    #store the fit with the best chisq and best Q for each eigenval, t0 bin
    Q_matrix = np.full( (16, 40, 12), np.nan, dtype=float)
    chi_sq_matrix = np.full( (16, 40, 12), np.nan, dtype=float)

    #solve the GEVP for each t0
    for t0 in range(t0_ind_start, t0_ind_end+1):

        GEVP_eigenvals = get_GEVP_eigenvals(t0)
        scaled_up_eigenvals = get_scaled_up_eigenvals(GEVP_eigenvals)
        avg, cov, inv_cov, std_err = get_eigenval_stats(scaled_up_eigenvals, t0)

        #fit every eigenval at this t0
        for eigen_ind in range(eigen_ind_start, eigen_ind_end+1):

            #choose the range of times within which to fit this eigenval
            #calculate the percent error at each time slice
            error_ratio = np.empty((40), dtype=float)
            for l in range(40):
                error_ratio[l] = std_err[eigen_ind,l]/avg[eigen_ind,l]

            #####################################################
            #to determine the fitting range, start at t0 and move outward in
            #both directions until percent errors get too big
            l=t0
            ratio = error_ratio[l]
            while ratio < 0.15 and l>0:
                l = l-1
                ratio = error_ratio[l]
            if ratio < 0.15:
                tmin_lim = l
            else:
                tmin_lim = l+1

            l=t0
            ratio = error_ratio[l]
            while ratio < 0.15 and l<39:
                l = l+1
                ratio = error_ratio[l]
            if ratio < 0.15:
```

```
            tmax_lim = l
        else:
            tmax_lim = l-1

        #if the fitting range is too narrow, the eigenval can't be fit
        #then break because all eigenvals need to be fit at the same t0
        if tmax_lim-tmin_lim<25 or tmin_lim>4:
            print("Errors too large at i="+str(eigen_ind)+", t0="+str(t0))
            break
        #####################################################

        if ( t0>t0_ind_start ) and \
        ( np.isnan(Q_matrix[eigen_ind,t0-1,4])==False ) and \
        Q_matrix[eigen_ind,t0-1,4]<2:
        #if the previous t0 gave a valid E1 and reasonble chisq,
        #use E1 guess from previous t0
            data = run_fits(avg[eigen_ind], inv_cov[eigen_ind], t0, \
            tmin_lim, tmax_lim, E1_guess=Q_matrix[eigen_ind,t0-1,8])
        else:
            data = run_fits(avg[eigen_ind], inv_cov[eigen_ind], t0, \
            tmin_lim, tmax_lim)

        #if data for this t0 is all nans
        #break because all eigenvals need to be fit at the same t0
        if np.isnan(np.nanmax(data[:,4]))==True:
            print("No good fits for i="+str(eigen_ind)+", t0="+str(t0))
            break

        else: #if the data isn't all nans, save to .csv
            save_str = "eigen_" + str(eigen_ind) + "_t0_" + str(t0) + ".csv"
            np.savetxt(save_str, data, delimiter=",")

            print("Ran fits for i="+str(eigen_ind)+", t0="+str(t0))

            #find the fits with best chisq and Q values
            chisq_per_ndof = data[:,4]
            Q = data[:,5]
            best_chisq_ind = np.nanargmin(chisq_per_ndof)
            best_Q_ind = np.nanargmax(Q)
            #save best fits in separate arrays
            Q_matrix[eigen_ind, t0] = data[best_Q_ind]
            chi_sq_matrix[eigen_ind, t0] = data[best_chisq_ind]

            #check for negative chisq
            min_chisq_per_ndof = np.nanmin(chisq_per_ndof)
            if min_chisq_per_ndof <= 0:
                print("Nonpositive chisq at eigenvalue " + str(eigen_ind)\
                 + ", t0="+str(t0))
                print("chisq="+str(best_chisq[4]))

#after iterating over all t0
```

```
    for eigen_ind in range(eigen_ind_start, eigen_ind_end+1):

        #save the best fit data
        np.savetxt("best_Q_per_t0_"+str(eigen_ind)+".csv", Q_matrix[eigen_ind], delimiter=",")
        np.savetxt("best_chisq_ndof_per_t0_"+str(eigen_ind)+".csv", chi_sq_matrix[eigen_ind], del
```

A.4 Utility Functions

The functions in this section perform simple operations and are included only so that the code is self-contained.

```
def symmetrize(array):
    """
    Symmetrize a matrix by averaging corresponding values.
    """
    #expects array of shape (dim(C), dim(C))
    dim = array.shape[0]
    for i in range(dim):
        for j in range(i,dim):
            array[i,j] = (array[i,j]+array[j,i])/2
            array[j,i] = array[i,j]
    return array
#______________________________________________________________________

def get_raw_data(filepath):
    """
    Iterates through the raw data and organizes it into a numpy array.
    filepath is the directory address for the datafiles.
    """
    ensemble_size = 469
    timeslices = 40
    dim = 16
    raw_array = np.empty([timeslices, ensemble_size, dim, dim], dtype=float)

    #Iterate through datafiles
    for i in range(dim):
        for j in range(dim):
            fullpath = filepath + "/C_" + str(i) + "_" + str(j)
            datafile = open(fullpath, "r")
            lines = datafile.readlines()

            for k in range(timeslices):
                for l in range(ensemble_size):
                    #line number for the kth timeslice in the lth ensemble member
                    line_number = 1 + k + 40*l
                    line = lines[line_number]
                    #find the index of the space, which is one index before the numerical data
                    space_ind = line.find(" ")
                    data = float(line[space_ind+1:])
```

```
                raw_array[k,l,i,j] = data

    return raw_array
#_______________________________________________________________
def make_avg_matrix(raw_array):
    """
    Returns the ensemble average matrix.
    """
    #expects array shape (times, configurations, dim(C), dim(C))
    array_shape = raw_array.shape
    timeslices = array_shape[0]
    ensemble_size = array_shape[1]
    dim = array_shape[2]

    avg_array = np.empty([timeslices, dim, dim], dtype=float)

    for i in range(dim):
        for j in range(dim):
            for k in range(timeslices):
                avg = 0.0
                for l in range(ensemble_size):
                    avg += raw_array[k,l,i,j]
                avg = avg/ensemble_size
                avg_array[k,i,j] = avg

    return avg_array
#__________________________________________________________
def jackkinfe_down(raw_val, mean, n):
    """
    Computes the jackknife scaled-down value.
    """
    scaled_down_val = mean-(raw_val-mean)/(n-1)
    return scaled_down_val
#______________________________________________________________________________________

def jackkinfe_up(raw_val, mean, n):
    """
     Computes the jackknife scaled-up value.
    """
    scaled_up_val = mean-(n-1)*(raw_val-mean)
    return scaled_up_val
#________________________________________
def scale_down_and_symmetrize(raw_array, avg_array):
    """
    Jackknife scales down the entries in each correlation matrix in the
    ensemble, then symmetrizes the matrices.
    """
    #expects an array of shape (times, configurations, dimc(C), dim(C))
    array_shape = raw_array.shape
    timeslices = array_shape[0]
    ensemble_size = array_shape[1]
```

```
    dim = array_shape[2]

    scaled_down_array = np.empty(array_shape, dtype=float)

    for i in range(dim):
        for j in range(dim):
            for k in range(timeslices):
                avg_val = avg_array[k,i,j]
                for l in range(ensemble_size):
                    raw_val = raw_array[k,l,i,j]
                    scaled_down_val = jackkinfe_down(raw_val, avg_val, ensemble_size)
                    scaled_down_array[k,l,i,j] = scaled_down_val

    #Symmetrize
    for k in range(timeslices):
        for l in range(ensemble_size):
            scaled_down_array[k,l] = symmetrize(scaled_down_array[k,l])

    return scaled_down_array
#_______________________________________________
def eigh_dec_order(matrix):
    """
    Returns the eigenvalues and eigenvectors of a Hermitian matrix in decreasing order.
    """
    eigenvals, vectors = np.linalg.eigh(matrix)
    eigenvals, vectors = reverse_eigenvector_order(eigenvals, vectors)

    return eigenvals, vectors
#____________________________________

def reverse_eigenvector_order(eigenvals, eigenvectors):
    """
    Returns the eigenvalues and eigenvectors in reverse order.
    """
    eigendim = len(eigenvals)

    reordered_eigenvals = np.empty(eigenvals.shape, dtype=float)
    reordered_vectors = np.empty(eigenvectors.shape, dtype=float)

    for j in range(eigendim):
        reordered_eigenvals[j] = eigenvals[eigendim-1-j]
        reordered_vectors[:,j] = eigenvectors[:,eigendim-1-j]

    return reordered_eigenvals, reordered_vectors
```

Bibliography

[1] Knechtli, Francesco; Günther, Michael and Peardon, Michael. Lattice Quantum Chromodynamics: Practical Essentials. Springer, 2017.

[2] D. Wilson, R. Briceño, J. Dudek, R. Edwards and C. Thomas, Phys. Rev. **D92** (2015) 094502.

[3] Gasiorowicz, Stephen. Quantum Physics. New York: Wiley, 2003.

[4] J. Dudek, R. Edwards, N. Mathur and D. Richards, Phys. Rev. **D77** (2008) 034501.

[5] B. Blossier, M. Della Morte, G. Von Hippel, T. Mendes and R. Sommer, JHEP **04** (2009) 094.

[6] Press, William H., Teukolsky, Saul A., Vetterling, William T. and Flannery, Brian P. Numerical Recipes in C. 2nd ed. Cambridge: Cambridge University Press, 1992.