ABSTRACT

Initial Systematic Investigations of the Weakly Coupled Free Fermionic Heterotic String Landscape Statistics

Timothy Renner, Ph.D.

Advisor: Gerald B. Cleaver, Ph.D.

A C++ framework was constructed with the explicit purpose of systematically generating string models using the Weakly Coupled Free Fermionic Heterotic String (WCFFHS) method. The software, optimized for speed, generality, and ease of use, has been used to conduct preliminary systematic investigations of WCFFHS vacua. Documentation for this framework is provided in the Appendix. After an introduction to theoretical and computational aspects of WCFFHS model building, a study of ten-dimensional WCFFHS models is presented. Degeneracies among equivalent expressions of each of the known models are investigated and classified. A study of more phenomenologically realistic four-dimensional models based on the well known "NAHE" set is then presented, with statistics being reported on gauge content, matter representations, and space-time supersymmetries. The final study is a parallel to the NAHE study in which a variation of the NAHE set is systematically extended and examined statistically. Special attention is paid to models with "mirroring" — identical observable and hidden sector gauge groups and matter representations.

Initial Systematic Investigations of the Weakly Coupled Free Fermionic Heterotic String Landscape Statistics

by

Timothy Renner, B.S.

A Dissertation

Approved by the Department of Physics

_____

Gregory A. Benesh, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Doctor of Philosophy

Approved by the Dissertation Committee

_____

Gerald B. Cleaver, Ph.D., Chairperson

_____

Jay R. Dittmann, Ph.D.

_____

Lorin S. Matthews, Ph.D.

_____

David J. Ryden, Ph.D.

_____

Anzhong Wang, Ph.D.

Accepted by the Graduate School
August 2011

_____

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

APPENDIX

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ST: Space-time.

WS: World Sheet.

SUSY: Supersymmetry.

GUT: Grand unified theory.

WCFFHS: Weakly Coupled Free Fermionic Heterotic String. A method of constructing heterotic string models.

VEV: Vacuum expectation value.

SU(N+1): Group of special unitary matrices of dimension $N+1$. Corresponds to the Cartan classification A_N.

SO(2N+1): Group of special orthogonal matrices of odd dimension $2N+1$. Corredsponds to the Cartan classification B_N.

Sp(2N): Group of symplectic matrices of dimension $2N$. Corresponds to the Cartan classification C_N.

SO(2N): Group of special orthogonal matrices of even dimension $2N$. Corresponds to the Cartan classification D_N.

$E_{6,7,8}, F_4, G_2$: Exceptional groups which only exist with certain dimensions. Their Cartan classification is their colloquial name.

KM: Kač-Moody.

GSOP: GSO (Gliozzi, Scherk, Olive) projection. In WCFFHS models, an equation each state in the model must satisfy for the Hilbert space of states to be invariant under modular transformations.

ABK: Antoniadis, Bachas, Kounnas. Group authoring one of the first WCFFHS model construction papers.

AB: Antoniandis, Bachas. Group which extended the work of ABK to include modes with any rational phase.

KLT: Kawai, Lewellen, Tye. Group authoring one of the first WCFFHS model construction papers.

NAHE: Nanopoulos, Antoniadis, Hagelin, Ellis. Group authoring the initial papers on quasi-realistic WCFFHS model building. Discovered a set of five order-2 basis vectors which serve as a basis from which phenomenologically realistic WCFFHS models are constructed.

ACKNOWLEDGMENTS

I would like first and foremost to thank my advisor, Professor Gerald Cleaver. His kind demeanor and cheerful disposition made even the toughest problems seem solvable. He has always let me address and solve the challenges thoughout my graduate career in my own way, lending a nearly inexhaustible breadth of knowledge to guide my next steps. For that, he has my utmost gratitude and respect.

I would like to thank all of the professors that have impacted my education through exemplary teaching: Professors Lorin Matthews, Yumei Wu, Bennie Ward, Gerald Cleaver, David Ryden, Anzhong Wang, Ken Park, and Jeff Olafsen. I would also like to thank the professors who have taken the time to serve on my defense committee: Professors Gerald Cleaver, Jay Dittmann, Lorin Matthews, Anzhong Wang, and David Ryden. I thank the Baylor Physics Department for allowing me to teach for them. My time teaching for the department has been excellent, and I am grateful to have had that opportunity. In particular I want to thank those who have supervised me as a teaching assistant: Drs. Tibra Ali and Ray Nazzario, Mr. Randy Hall, and especially Dr. Linda Kinslow. Finally, I thank Mrs. Marian Nunn-Graves and Mrs. Chava Baker for all of their assistance.

I owe a great deal to the members, past and present, of the EUCOS group: Jared Greenwald, Kristen Pechan, Erik Remkus, Yanbin Deng, and especially Doug Moore. Doug has been a seemingly endless font of knowledge regarding nearly every question I can think to ask. His assistance has been crucial both to the EUCOS group, as well as my own research. I owe him a great debt.

I would like to thank the String Vacuum Project for funding my travel to their conferences and allowing me to present my work there.

I would also like to thank my family for their constant love and support. My father Steve has taught me that there are a great many valuable things in life that

cannot be quantified. He fostered in me a love of literature and poetry that has both guided me and echoed throughout my scientific endeavors. My mother Alice has been a model of strength and persistence. Through a great many difficulties — lupus, back surgery, breast cancer, and (possibly the most difficult) life with my father — she has always remained a constant source of laughter and encouragement. I would like to thank my sister Sara, as she has been a model of excellence in her academic and spiritual pursuits. Giving up is a thought she has clearly never had, and I will always benefit from that example. Altogether my family has taught me a skill that will continue to make my life wonderful: laughter. Never unwilling to miss a chance at laughter, and never unwilling to let a humbling moment escape my attention, they have shown me that the best way to enjoy life is to laugh, especially at myself.

My friends have always been a source of strength and encouragement, as well as an excellent source of knowledge (most of the time). In particular, my "formative" years were shaped by the presence of Drs. Andreas Tziolas and Richard Obousy, for better or worse. They made me feel welcome during the exceedingly difficult first years of graduate school, making the transition here much easier. I also want to thank Jay Murphree, BJ Enzweiler, Jonathan Perry, Dr. Victor Land, Anne Land-Zandstra, Dr. Victor Guerrero, Dr. Sammy Joseph, Doug Moore, and V.H. Satheeshkumar for their continuing support. My friends outside the department also have my thanks — Andy and Tamee Ryan, Norbert Trimmer, Amanda Piccolo, Sarah Hutchins, Rev. Dr. Robert Flowers, and Jojo Percy.

The members of the Central Christian Church Chancel Choir also have my gratitude. My Wednesday nights have been filled not only with laughter, but also with reminders to call my mother. I thank Sarah Daniels and Chris Diamond for allowing me to sing with such a wonderful, fun group of people.

I also want to acknowledge my two closest friends in graduate school: Dr. Martin Frank and James Creel. The trials, tribulations, and discussions Martin and I have been part of would make an outstanding piece of epic poetry. I am convinced that with enough malted beverages and chicken wings, Martin and I could solve a great deal of the world's problems. James has been a close friend and confidant throughout my time here at Baylor. His generosity and loyalty are something for which I will always be grateful.

Finally, I would like to thank my partner, Cindy Calvert. Cindy has been by my side through most of the graduate school process, and her love has kept me going through the most difficult parts of it. Her understanding and patience are traits that I truly admire. She has put aside a great deal of her personal goals to allow me to complete graduate school, and I hope to repay that debt someday.

It is to all of my friends that I dedicate this dissertation.

For my friends

CHAPTER ONE

Introduction

1.1 String Theory

Currently the Standard Model (SM) of particle physics describes the universe up to the electroweak scale $M_W = 246$ GeV [1], but has several shortcomings. Firstly, the interactions of the fundamental particles described by the symmetry groups $SU(3)_C \otimes SU(2)_L \otimes U(1)_Y$ where C, L, and Y denote the color force, weak isospin, and electroweak hypercharge, respectively, are only valid when the gauge bosons are massless. At low energies $W^{\pm}$ and Z^0, the electroweak bosons, become massive. This discrepancy is solved via the Higgs mechanism through a process called spontaneous symmetry breaking. Such a mechanism introduces new scalar bosons (denoted Higgs bosons), which have yet to be observed experimentally.

Additionally, the SM coupling constants for the three forces renormalize[1] at high energies to nearly the same value, but do not all intersect at one point. Such an intersection would mean that the fundamental forces are different manifestations of the same unified force. Introducing supersymmetry (SUSY) allows this to happen. SUSY is a symmetry in which each known fermion has a bosonic superpartner, and each known boson has a fermion superpartner. Such a mechanism forces cancellations to occur in the renormalization of the couplings that allows all three couplings to approach one common value around 2.5×10^{16} GeV.

The SM also requires around twenty free parameters to describe itself, many of which must be fine tuned. This lack of simplicity suggests that the SM is a low energy projection of a higher energy theory with fewer free parameters.

[1] Renormalization is a process that stabilizes the UV limits of interaction cross sections. Couplings, masses, and the fields themselves all must be renormalized to produce mathematically consistent results.

Perhaps the greatest shortcoming of the SM is the lack of a gravitational force. The mathematical framework of the Standard Model, called quantum field theory (QFT), cannot properly describe the gravitational interaction. In contrast to the SM forces, renormalizing gravity results is an infinite cross section that must be fine tuned at every interaction order to be finite.

A more encompassing description must therefore be explored to fully explain the universe at the fundamental level, and one such option is string theory. String theory describes the fundamental interactions of physics not with point particles, as QFT does, but with one-dimensional objects denoted as "strings." A string with certain (quantum mechanical) vibration modes will have a low energy projection of one particle while a string with different vibration modes will produce a completely different particle. Thus, rather than a theory with a large number of fundamentally different particles, string theory uses a single object (the string) vibrating at different modes to describe the universe. While there are some aspects of string theory that may seem initially unappealing (to be described later), many of these can be used in a way that produces a supersymmetric, unified description of the universe.

1.1.1 Bosonic Strings

The first string theory put forward was an initial attempt to explain the symmetry of the strong nuclear force, but was found to be inferior to quantum chromodynamics (QCD). It became of interest again when it was realized that certain spin-2 closed string states that could not be eliminated from the theory described quantum mechanical carriers of the gravitational force, gravitons. Thus, this first attempt at string theory necessarily included quantum gravity. Unfortunately, this theory lacked a crucial component of the universe: matter. There are no space-time fermion modes in what became known as bosonic string theory, and thus this framework cannot be used to fully describe the universe. Nevertheless, bosonic strings

are used as a component in string theories with space-time fermions (specifically, the heterotic string theories), and should be discussed. The discussion to follow summarizes the approach in refs. [2, 3].

A classical bosonic string is described by the Polyakov action.

$$S_P = \frac{1}{4\pi\alpha'}\int_\Sigma d^2\xi\sqrt{h}h^{ab}\partial_a X^\mu \partial_b X_\mu, \tag{1.1}$$

where Σ is the two-dimensional surface a string creates as it moves through space-time, or world-sheet, ξ^0, ξ^1 are world sheet coordinates, X_μ are the Minkowski space-time coordinates of the string, α' is the string tension, and h^{ab} is the intrinsic world sheet metric. The Euler-Lagrange equations yield the following equation of motion.

$$\partial_\alpha(\sqrt{-h}h^{\alpha\beta}\partial_\beta X^\mu) = 0 \tag{1.2}$$

These equations are solved by the following mode expansions:

$$X^\mu = X^\mu_L + X^\mu_R \tag{1.3}$$

where

$$X^\mu_L = \frac{1}{2}x^\mu + \alpha' p^\mu(\xi^0 - \xi^1) + i\sqrt{\frac{\alpha'}{2}}\sum_{n\neq 0}\frac{1}{n}\alpha^\mu_n e^{-2in(\xi^0-\xi^1)} \tag{1.4}$$

and

$$X^\mu_R = \frac{1}{2}x^\mu + \alpha' p^\mu(\xi^0 + \xi^1) + i\sqrt{\frac{\alpha'}{2}}\sum_{n\neq 0}\frac{1}{n}\overline{\alpha}^\mu_n e^{-2in(\xi^0+\xi^1)} \tag{1.5}$$

Thus,

$$X^\mu = x^\mu + 2\alpha' p^\mu \xi^0 + i\sqrt{\frac{\alpha'}{2}}\sum_{n\neq 0}\frac{1}{n}e^{-2in\xi^0}(\alpha^\mu_n e^{2in\xi^1} + \overline{\alpha}^\mu_n e^{-2in\xi^1}) \tag{1.6}$$

Varying the Polyakov action with respect to $\dot{X}_\mu$ yields the conjugate momentum P^μ.

$$P^\mu = \frac{1}{2\pi\alpha'}\frac{\partial X^\mu}{\partial\xi^0} = \frac{p^\mu}{\pi} + \frac{1}{\pi\sqrt{2\alpha'}}\sum_{n\neq 0}e^{-2in\xi^0}(\alpha^\mu_n e^{2i\xi^1} + \overline{\alpha}^\mu_n e^{-2i\xi^1}). \tag{1.7}$$

With expressions for the conjugate momentum p^μ and the field X^μ in place, one can write the commutation relations in anticipation of quantization

$$[P^\mu(\xi^1,\xi^0), P^\nu(\xi^{1'},\xi^0)] = [X^\mu(\xi^1,\xi^0), X^\nu(\xi^{1'},\xi^0)] = 0, \tag{1.8}$$

$$[P^\mu(\xi^1,\xi^0), X^\nu(\xi^{1'},\xi^0)] = i\eta^{\mu\nu}\delta(\xi^1-\xi^{1'}). \tag{1.9}$$

Rewriting this in terms of the excitation mode operators, $\alpha_n^\mu,\ \overline{\alpha}_n^\mu$,

$$[\alpha_n^\mu, \overline{\alpha}_m^\nu] = 0, \tag{1.10}$$

$$[\alpha_n^\mu, \alpha_m^\nu] = [\overline{\alpha}_n^\mu, \overline{\alpha}_m^\nu] = m\eta^{\mu\nu}\delta_{m+n,0}. \tag{1.11}$$

Redefining the excitation mode operators allows them to be expressed as a raising/lowering algebra for harmonic oscillators...

$$a_m^\mu = \frac{1}{\sqrt{m}}\alpha_m^\mu, \tag{1.12}$$

$$a_m^{\mu\dagger} = \frac{1}{\sqrt{m}}\alpha_{-m}^\mu. \tag{1.13}$$

The commutation relations (1.11) become

$$[a_m^\mu, a_n^{\nu\dagger}] = [\overline{a}_m^\mu, \overline{a}_n^{\nu\dagger}] = \eta^{\mu\nu}\delta_{mn} \tag{1.14}$$

for $m, n > 0$. An expression for the Hamiltonian that will be useful for future calculations is

$$H = \int_0^\pi (\dot{X}_\mu P_0^\mu - \mathcal{L}) d\xi^1. \tag{1.15}$$

Inserting the mode expansion (1.6) results in

$$H = \sum_{n=-\infty}^{\infty} (\alpha_{-n}\cdot\alpha_n + \overline{\alpha}_{-n}\cdot\overline{\alpha}_n). \tag{1.16}$$

Varying the Polyakov action with respect to the intrinsic metric h yields the following equations for the energy-momentum tensor:

$$T_{\alpha\beta} = \partial_\alpha X^\mu \partial_\beta X_\mu - \frac{1}{2}h_{\alpha\beta}h^{\gamma\delta}\partial_\gamma X^\mu \partial_\delta X_\mu = 0, \tag{1.17}$$

where $h_{\alpha\beta}$ is the intrinsic metric. These equations are better expressed and solved in light cone coordinates, defined by

$$\xi^\pm = \xi^0 \pm \xi^1. \tag{1.18}$$

The wave equation for the string (1.2) becomes

$$\partial_+\partial_- X^\mu = 0. \tag{1.19}$$

The closed string mode expansions (1.4, 1.5) become

$$X_L = \frac{1}{2}x^\mu + \alpha' p^\mu \xi^- + i\sqrt{\frac{\alpha'}{2}}\sum_{n\neq 0}\frac{1}{n}\alpha_n^\mu e^{-2in\xi^-}, \tag{1.20}$$

$$X_R = \frac{1}{2}x^\mu + \alpha' p^\mu \xi^+ + i\sqrt{\frac{\alpha'}{2}}\sum_{n\neq 0}\frac{1}{n}\overline{\alpha}_n^\mu e^{-2in\xi^+}. \tag{1.21}$$

The equations (1.17) for the energy-momentum tensor then become

$$T_{++} = \partial_+ X^\mu \partial_+ X_\mu = 0, \tag{1.22}$$

$$T_{--} = \partial_- X^\mu \partial_- X_\mu = 0. \tag{1.23}$$

Placing the mode expansions (1.20, 1.21) into (1.22, 1.23) gives the following equations for the energy-momentum tensor:

$$T_{--} = 2\alpha' \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty}\alpha_{m-n}\cdot\alpha_n e^{-2im\xi^-} = 4\alpha'\sum_{m=-\infty}^{\infty} L_m e^{-2im\xi^-} = 0, \tag{1.24}$$

$$T_{++} = 2\alpha' \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty}\overline{\alpha}_{m-n}\cdot\overline{\alpha}_n e^{-2im\xi^+} = 4\alpha'\sum_{m=-\infty}^{\infty} \overline{L}_m e^{-2im\xi^+} = 0 \tag{1.25}$$

$L_m,\ \overline{L}_m$ are called Virasoro operators. They are defined as follows:

$$L_m = \frac{1}{2}\sum_{n=-\infty}^{\infty}\alpha_{m-n}\cdot\alpha_n \tag{1.26}$$

$$\overline{L}_m = \frac{1}{2}\sum_{n=-\infty}^{\infty}\overline{\alpha}_{m-n}\cdot\overline{\alpha}_n. \tag{1.27}$$

Quantum mechanically, to eliminate infinities, normal ordering must be imposed on the Virasoro operators. This means that the lowering operators appear to the right of the raising operators.

$$:L_m: = \frac{1}{2}\sum_{n=-\infty}^{\infty}:\alpha_{m-n}\cdot\alpha_n: \tag{1.28}$$

$$:\overline{L}_m: = \frac{1}{2}\sum_{n=-\infty}^{\infty}:\overline{\alpha}_{m-n}\cdot\overline{\alpha}_n: \tag{1.29}$$

The only Virasoro operators affected by the ordering prescription are L_0 and $\overline{L}_0$. This is due to the commutation relations for the α and $\overline{\alpha}$ operators. Explicitly,

$$:L_0: = \frac{1}{2}\alpha_0^2 + \sum_{n=1}^{\infty} \alpha_{-n} \cdot \alpha_n \tag{1.30}$$

$$:\overline{L}_0: = \frac{1}{2}\overline{\alpha}_0^2 + \sum_{n=1}^{\infty} \overline{\alpha}_{-n} \cdot \overline{\alpha}_n \tag{1.31}$$

A constant is needed to encode the arbitrary nature of the normal ordering. In general,

$$:L_0: = L_0 + a, \tag{1.32}$$

$$:\overline{L}_0: = \overline{L}_0 + a, \tag{1.33}$$

where $:L_0:$ and $:\overline{L}_0:$ are the quantum mechanical Virasoro operators, L_0 and $\overline{L}_0$ are classical, and a is the normal ordering constant, which will be solved for later.

The Hamiltonian (1.16) can be expressed in terms of the Virasoro operators.

$$H = \sum_{n=-\infty}^{\infty} \alpha_{-n} \cdot \alpha_n + \overline{\alpha}_{-n} \cdot \overline{\alpha}_n = 2(:L_0: + :\overline{L}_0:) \tag{1.34}$$

The commutation relations for the Virasoro operators can be determined using the commutators for the α operators. They are

$$[L_m, L_n] = (m-n)L_{m+n} + \frac{D}{12}m(m^2-1)\delta_{m+n,0}, \tag{1.35}$$

$$[\overline{L}_m, \overline{L}_n] = (m-n)\overline{L}_{m+n} + \frac{D}{12}m(m^2-1)\delta_{m+n,0}, \tag{1.36}$$

where D is the number of space-time dimensions. The term involving D is called a *central extension* to the Virasoro algebra, and is a quantum mechanical correction of the form $A(m)\delta_{m+n}$. Solving for $A(m)$ involves using the Jacobi identity to develop a recursion relation, then using the expectation value of the commutators.

The Virasoro operators can also be used to derive the mass shell condition for the string. Starting with the relativistic mass expression,

$$M^2 = -p^\mu p_\mu, \tag{1.37}$$

the total momentum of the string is

$$p^\mu = \int_0^\pi d\xi^1 \dot{X}^\mu(\xi^1). \tag{1.38}$$

Due to the bounds of this integral, only the zero modes contribute. This implies

$$p^2 = \frac{1}{\alpha'}(\alpha_0^2 + \overline{\alpha}_0^2). \tag{1.39}$$

Thus, the Virasoro operator expression for the zero modes (1.30, 1.31) using the energy-momentum conditions (1.25, 1.24) is

$$:L_0: + :\overline{L}_0: = \sum_{n=1}^{\infty}(\alpha_{-n}\cdot\alpha_n + \overline{\alpha}_{-n}\cdot\overline{\alpha}_n) + \frac{\alpha'}{2}p^2 + 2a = 0. \tag{1.40}$$

Substituting the momentum term in the above equation with the relativistic momentum (1.37), then solving for the mass, results in

$$M^2 = \frac{2}{\alpha'}\sum_{n=1}^{\infty}(\alpha_{-n}\cdot\alpha_n + \overline{\alpha}_{-n}\cdot\overline{\alpha}_n) - 2a. \tag{1.41}$$

Altogether, these equations give a mass shell condition for the string,

$$:L_0:|\phi\rangle = (L_0 - a)|\phi\rangle = 0, \tag{1.42}$$

$$:\overline{L}_0:|\phi\rangle = (\overline{L}_0 - a)|\phi\rangle = 0, \tag{1.43}$$

where $|\phi\rangle$ is any physical on-shell state in the theory. Additionally, the mass-shell condition imposes a constraint between the left and right moving zero mode Virasoro operators.

$$(:L_0: - :\overline{L}_0:)|\phi\rangle = (L_0 - \overline{L}_0)|\phi\rangle = 0. \tag{1.44}$$

This is known as the level matching condition.

In the light cone gauge, it can be shown that the values of D, the space-time dimension, and a, the normal ordering constant, cannot be arbitrary. This is a result of maintaining Lorentz invariance. Begin by writing the mode expansions of

the (normal ordered) L_0 and $\overline{L}_0$ operators in the light cone gauge:

$$L_0 = \frac{1}{2}\alpha_0^2 + \frac{1}{2}\sum_{i=1}^{D-2}\sum_{n\neq 0}\alpha_{-n}^i\alpha_n^i, \tag{1.45}$$

$$\overline{L}_0 = \frac{1}{2}\overline{\alpha}_0^2 + \frac{1}{2}\sum_{i=1}^{D-2}\sum_{n\neq 0}\overline{\alpha}_{-n}^1\overline{\alpha}_n^i. \tag{1.46}$$

These expansions are not yet normal ordered. Imposing normal ordering leads to the following expressions

$$:L_0: = \frac{1}{2}\alpha_0^2 + \sum_{i=1}^{D-2}\sum_{n=1}^{\infty}\alpha_{-n}^i\alpha_n^i + (\frac{D-2}{2})\sum_{n=1}^{\infty} n, \tag{1.47}$$

$$:\overline{L}_0: = \frac{1}{2}\overline{\alpha}_0^2 + \sum_{i=1}^{D-2}\sum_{n=1}^{\infty}\overline{\alpha}_{-n}^i\overline{\alpha}_n^i + (\frac{D-2}{2})\sum_{n=1}^{\infty} n. \tag{1.48}$$

Using the identity

$$\sum_{n=1}^{\infty} n = \zeta(-1) = -\frac{1}{12}, \tag{1.49}$$

where $\zeta(n)$ is the Riemann zeta function, the following expressions are produced for the Virasoro operators

$$:L_0: = \frac{1}{2}\alpha_0^2 + \sum_{i=1}^{D-2}\sum_{n=1}^{\infty}\alpha_{-n}^i\alpha_n^i - \frac{D-2}{24}, \tag{1.50}$$

$$:\overline{L}_0: = \frac{1}{2}\overline{\alpha}_0^2 + \sum_{i=1}^{D-2}\sum_{n=1}^{\infty}\overline{\alpha}_{-n}^i\overline{\alpha}_n^i - \frac{D-2}{24}. \tag{1.51}$$

The normal ordering constant a can now be written as a function of D.

$$a = \frac{D-2}{24} \tag{1.52}$$

Additionally, the expressions for the normal ordered operators defined by (1.50, 1.51) can be used to solve for the mass. Firstly, compute $L_0 + \overline{L}_0$,

$$:L_0: + :\overline{L}_0: = -\frac{\alpha'}{2}M^2 + \sum_{i=1}^{D-2}\sum_{n=1}^{\infty}(\alpha_{-n}^i\alpha_n^i + \overline{\alpha}_{-n}^i\overline{\alpha}_n^i) - \frac{D-2}{12} = 0. \tag{1.53}$$

Defining the number operators, N and $\overline{N}$, which count the number of excitations for a given state, as follows

$$N = \sum_{i=1}^{D-2} \sum_{n=1}^{\infty} \alpha_{-n}^{i} \alpha_{n}^{i}, \tag{1.54}$$

$$\overline{N} = \sum_{i=1}^{D-2} \sum_{n=1}^{\infty} \overline{\alpha}_{-n}^{i} \overline{\alpha}_{n}^{i}, \tag{1.55}$$

leads to a simpler expression for the mass-shell of the string:

$$:L_0: + :\overline{L}_0:= -\frac{\alpha'}{2} M^2 + N + \overline{N} - \frac{D-2}{12} = 0. \tag{1.56}$$

Solved for the mass, this expression relates the mass of the string to the number of excited states N and $\overline{N}$ and the number of space-time dimensions D:

$$\frac{\alpha'}{2} M^2 = N + \overline{N} - \frac{D-2}{12}. \tag{1.57}$$

Note that the level matching condition (1.44) becomes, in terms of the number operators

$$L_0 = \overline{L_0} \rightarrow N = \overline{N}. \tag{1.58}$$

This means that there must be the same number of excitations on the right as there are on the left. For $N = \overline{N} = 0$, the mass formula (1.57) gives a negative mass squared, so that state is tachyonic. The next excited state, $N = \overline{N} = 1$, is massless by Lorentz invariance. The space-time dimension D and the normal ordering constant a can now be solved for

$$D = 26, \tag{1.59}$$

$$a = 1. \tag{1.60}$$

26 is considered to be the *critical dimension* for bosonic string theory, which leaves it at a disadvantage, as there are only four observed space-time dimensions. As will be shown shortly, introducing supersymmetry reduces the number of required space-time dimensions to 10.

1.1.2 Superstrings

The only currently known way to produce a massless ST fermion spectrum in string theory is to impose a world sheet supersymmetry on the action. The discussion to follow summarizes the approach in refs. [4, 5]. For a curved background, the generalized supersymmetric Polyakov action is

$$S = \frac{1}{4\pi\alpha'}\int d^2\xi\sqrt{-\gamma}\left[\gamma^{ab}\partial_a X^\mu \partial_b X^\nu + i\bar{\psi}^\mu\gamma^a\nabla_a\psi^\nu \right. \\ \left. -2\bar{\chi}_a\gamma^b\gamma^a\psi^\mu\partial_b X^\nu - \frac{1}{2}\bar{\psi}^\mu\psi^\nu\bar{\chi}_a\gamma^b\gamma^a\chi_b\right]\eta_{\mu\nu}, \tag{1.61}$$

where ξ is the world sheet coordinate, γ^{ab} is the world sheet metric, X^μ, X^ν are real boson fields, $\bar{\psi}^\mu, \psi^\nu$ are real fermion fields, γ^a is a two-dimensional Dirac matrix defined on a curved space, ∇_a is a covariant spinor derivative, $\bar{\chi}^a$ is a vector-spinor "gravitino" field needed to ensure local supersymmetry, and $\eta_{\mu\nu}$ is the space-time Minkowski metric. Local symmetries of this action allow a gauge to be chosen in which the following are true:

$$\gamma_{ab} = \eta_{ab}, \tag{1.62}$$

$$\chi_a = 0, \tag{1.63}$$

where η_{ab} is the flat world sheet metric. The action (1.61) then becomes the Ramond-Neveu-Schwarz (RNS) superstring action, defined by

$$S = \frac{1}{4\pi\alpha'}\int d^2\xi\eta^{ab}(\partial_a X^\mu\partial_b X^\nu + i\bar{\psi}^\mu\gamma_a\partial_b\psi^\nu)\eta_{\mu\nu}. \tag{1.64}$$

The first term is a kinetic term for the boson fields, and is identical to the kinetic term in the bosonic string action, (1.1). The second term is a kinetic term for the fermion fields, $\bar{\psi}^\mu, \psi^\nu$. The equations of motion for (1.61) are also affected by choosing the conformal gauge. Varying the generalized SUSY action, then imposing the gauge symmetry results in the following equations of motion for the boson and

fermion fields,

$$\eta^{ab}\partial_a\partial_b X^\mu = 0, \tag{1.65}$$

$$\eta^{ab}\gamma_a\partial_b\psi^\mu = 0. \tag{1.66}$$

Additionally, varying the action with respect to the $\bar{\chi}_a$ and γ^a fields result in constraints on the energy momentum tensor T_{ab} and world-sheet supercurrent J^a:

$$T_{ab} = \partial_a X^\mu \partial_b X_\mu + \frac{i}{2}\bar{\psi}^\mu\gamma_{(a}\partial_{b)}\psi^\mu - \frac{1}{2}\gamma_{ab}(\partial_i X^\mu \partial^i X_\mu + \frac{i}{2}\bar{\psi}^\mu\gamma_i\partial^i\psi^\mu) = 0, \tag{1.67}$$

$$J^a = \gamma^b\gamma^a\psi^\mu\partial_b X_\mu = 0. \tag{1.68}$$

The formalism can be greatly simplified by introducing light cone coordinates defined by equation (1.18), as was done with the bosonic string. It is also convenient to define Majorana-Weyl spinors for the fermion fields.

$$\frac{1}{2}(1+\gamma^3)\psi^\mu = \begin{pmatrix} \psi_-^\mu \\ 0 \end{pmatrix}, \tag{1.69}$$

$$\frac{1}{2}(1-\gamma^3)\psi^\mu = \begin{pmatrix} 0 \\ \psi_+^\mu \end{pmatrix} \tag{1.70}$$

where γ^3 is defined as a chirality operator such that

$$\gamma^3 = \gamma^0\gamma^1 \tag{1.71}$$

and the following equations are satisfied

$$\gamma^3\psi_\pm^\mu = \pm\psi_\pm^\mu. \tag{1.72}$$

The chirality states correspond now to handedness, so ψ_- is a left moving mode and ψ_+ is a right moving mode.

Rewriting the RNS string action (1.64) in this coordinate system results in

$$S = \frac{1}{4\pi\alpha'}\int d^2\xi(\partial_+ X^\mu\partial_- X_\mu + \frac{i}{2}\psi_+^\mu\partial_-\psi_{+\mu} + \frac{i}{2}\psi_-^\mu\partial_+\psi_{-\mu}). \tag{1.73}$$

The equations of motion for the bosonic fields are identical to the nonsupersymmetric boson field equations of motion (1.19). The fermion fields obey the equations

$$\partial_- \psi_+^\mu = 0, \tag{1.74}$$

$$\partial_+ \psi_-^\mu = 0, \tag{1.75}$$

and the energy-momentum tensor components become

$$T_{++} = \partial_+ X^\mu \partial_+ X_\mu + \frac{i}{2} \psi_+^\mu \partial_+ \psi_{+\mu} = 0, \tag{1.76}$$

$$T_{--} = \partial_- X^\mu \partial_- X_\mu + \frac{i}{2} \psi_-^\mu \partial_- \psi_{-\mu} = 0. \tag{1.77}$$

The supercurrent condition (1.68) becomes

$$J_+ = \psi_+^\mu \partial_+ X_\mu = 0, \tag{1.78}$$

$$J_- = \psi_-^\mu \partial_- X_\mu = 0. \tag{1.79}$$

The condition imposed when varying the action with respect to the fermion fields can be satisfied with two separate worldsheet boundary conditions: periodic boundary conditions (referred to as Ramond boundary conditions) and antiperiodic boundary conditions (referred to as Neveu-Schwarz boundary conditions). Closed superstrings, then, can have all possible combinations of Ramond (R) and Neveu-Schwarz (NS) boundary conditions for the left and right moving modes. The mode expansion solutions to the equations of motion (1.74, 1.75) for each of the modes and boundary conditions are as follows

$$\psi_-^\mu = \sqrt{\alpha'} \sum_{n=-\infty}^{\infty} d_n^\mu e^{-2in(\tau-\sigma)} \qquad R, \tag{1.80}$$

$$\psi_+^\mu = \sqrt{\alpha'} \sum_{n=-\infty}^{\infty} \overline{d}_n^\mu e^{-2in(\tau+\sigma)} \qquad \overline{R}, \tag{1.81}$$

$$\psi_-^\mu = \sqrt{\alpha'} \sum_{n=-\infty}^{\infty} b_{n+\frac{1}{2}}^\mu e^{-2i(n+\frac{1}{2})(\tau-\sigma)} \qquad NS, \tag{1.82}$$

$$\psi_+^\mu = \sqrt{\alpha'} \sum_{n=-\infty}^{\infty} \overline{b}_{n+\frac{1}{2}}^\mu e^{-2i(n+\frac{1}{2})(\tau+\sigma)} \qquad \overline{NS}, \tag{1.83}$$

where $d^{\mu}_{n}, b^{\mu}_{n+\frac{1}{2}}$ are Fourier coefficients for a left mover and $\overline{d}^{\mu}_{n}, \overline{b}^{\mu}_{n+\frac{1}{2}}$ are Fourier coefficients for a right mover.

To impose the conditions (1.76, 1.77) it is convenient, as was the case with the bosonic string, to introduce Virasoro operators, defined here by

$$L_m = L_m^X + L_m^{\psi}, \tag{1.84}$$

$$\overline{L}_m = \overline{L}_m^X + \overline{L}_m^{\psi}, \tag{1.85}$$

where the superscript indicates the bosonic or fermionic part of the operator. The bosonic part of the operator is identical to the Virasoro operator of the bosonic string for left and right moving fields, (1.26, 1.27). As with the bosonic string, the chief concern in using these operators is the derivation of the quantum theory's mass shell condition, which is determined from the normal ordering constant imposed on the L_0 and $\overline{L}_0$ operators. The fermion field L_0 and $\overline{L}_0$ operators, expressed in terms of their respective Fourier coefficients, are as follows

$$L_0^{\psi,R} = \frac{1}{2}\sum_{n=-\infty}^{\infty} n d^{\mu}_{-n} d_{n,\mu}, \tag{1.86}$$

$$\overline{L}_0^{\psi,R} = \frac{1}{2}\sum_{n=-\infty}^{\infty} n \overline{d}^{\mu}_{-n} \overline{d}_{n,\mu}, \tag{1.87}$$

$$L_0^{\psi,NS} = \frac{1}{2}\sum_{n=-\infty}^{\infty} \left(n+\frac{1}{2}\right) b^{\mu}_{-n-\frac{1}{2}} b_{n+\frac{1}{2},\mu}, \tag{1.88}$$

$$\overline{L}_0^{\psi,NS} = \frac{1}{2}\sum_{n=-\infty}^{\infty} \left(n+\frac{1}{2}\right) \overline{b}^{\mu}_{-n-\frac{1}{2}} \overline{b}_{n+\frac{1}{2},\mu}. \tag{1.89}$$

To quantize this theory, the same normal ordering condition as the bosonic string is applied, leading to the mass shell conditions

$$(L_0 - \delta)|\psi\rangle = 0, \tag{1.90}$$

$$(\overline{L}_0 - \overline{\delta})|\psi\rangle = 0. \tag{1.91}$$

For the bosonic string, the α_m^+ and $\overline{\alpha}_m^+$ were set to zero via the light cone gauge. Similarly, the longitudinal raising and lowering operators for fermions can also be set to zero:

$$d_m^+ = 0, \tag{1.92}$$

$$\overline{d}_m^+ = 0, \tag{1.93}$$

$$b_{n+\frac{1}{2}}^+ = 0, \tag{1.94}$$

$$\overline{b}_{n+\frac{1}{2}}^+ = 0. \tag{1.95}$$

For the bosonic ladder operators, the commutation relations (1.11) are imposed. For the fermionic ladder operators, anticommutation relations must be imposed

$$\{d_m^\mu, d_n^\nu\} = -\eta^{\mu\nu}\delta_{m+n,0}, \tag{1.96}$$

$$\left\{\overline{d}_m^\mu, \overline{d}_n^\nu\right\} = -\eta^{\mu\nu}\delta_{m+n,0}, \tag{1.97}$$

$$\left\{b_{m+\frac{1}{2}}^\mu, b_{n+\frac{1}{2}}^\nu\right\} = -\eta^{\mu\nu}\delta_{m+n+1,0}, \tag{1.98}$$

$$\left\{\overline{b}_{m+\frac{1}{2}}^\mu, \overline{b}_{n+\frac{1}{2}}^\nu\right\} = -\eta^{\mu\nu}\delta_{m+n+1,0}. \tag{1.99}$$

These anticommutation relations can be used to rewrite the zero mode fermionic Virasoro operators defined by equations (1.86, 1.87, 1.88, 1.89) in terms of number operators. The number operators are defined as follows:

$$N_R^\psi = \sum_{n=0}^{\infty} n d_{-n}^i d_n^i, \tag{1.100}$$

$$\overline{N}_R^\psi = \sum_{n=0}^{\infty} n \overline{d}_{-n}^i \overline{d}_n^i, \tag{1.101}$$

$$N_{NS}^\psi = \sum_{n+\frac{1}{2}=0}^{\infty} \left(n + \frac{1}{2}\right) b_{-n-\frac{1}{2}}^i b_{n+\frac{1}{2}}^i, \tag{1.102}$$

$$\overline{N}_{NS}^\psi = \sum_{n+\frac{1}{2}=0}^{\infty} \left(n + \frac{1}{2}\right) \overline{b}_{-n-\frac{1}{2}}^i \overline{b}_{n+\frac{1}{2}=0}^i, \tag{1.103}$$

where the index i goes only over the transverse directions of space-time. Notice that for nonzero values, the eigenvalues for NS modes are half integer. In terms of the

number operators, the zero mode fermion Virasoro operators become

$$L_0^{\psi,R} = -N_R^{\psi} - \frac{D-2}{24}, \tag{1.104}$$

$$\overline{L}_0^{\psi,R} = -\overline{N}_R^{\psi} - \frac{D-2}{24}, \tag{1.105}$$

$$L_0^{\psi,NS} = -N_{NS}^{\psi} + \frac{D-2}{48}, \tag{1.106}$$

$$\overline{L}_0^{\psi,NS} = -\overline{N}_{NS}^{\psi} + \frac{D-2}{48}, \tag{1.107}$$

where D is the number of space-time dimensions. From these newly expressed operators as well as the zero mode Virasoro operators for the bosonic fields, the mass shell conditions and "level matching" conditions for a closed superstring can be derived. For the $R - \overline{R}$ superstring, these conditions are

$$\alpha' M^2 = 2(N^X + N_R^{\psi} + \overline{N}^X + \overline{N}_R^{\psi}), \tag{1.108}$$

$$N^X + N_R^{\psi} = \overline{N}^X + \overline{N}_R^{\psi}, \tag{1.109}$$

where N^X and $\overline{N}^X$ are the left and right moving bosonic number operators, defined in equations (1.54, 1.55). For the $NS - \overline{NS}$ superstring,

$$\alpha' M^2 = 2\left(N^X + N_{NS}^{\psi} + \overline{N}^X + \overline{N}_{NS}^{\psi} - \frac{D-2}{8}\right), \tag{1.110}$$

$$N^X + N_{NS}^{\psi} = \overline{N}^X + \overline{N}_{NS}^{\psi}. \tag{1.111}$$

For the $R - \overline{NS}$ superstring,

$$\alpha' M^2 = 2\left(N^X + N_R^{\psi} + \overline{N}^X + \overline{N}_{NS}^{\psi} - \frac{D-2}{16}\right), \tag{1.112}$$

$$N^X + N_R^{\psi} = \overline{N}^X + \overline{N}_{NS}^{\psi} - \frac{D-2}{16}. \tag{1.113}$$

Finally, for the $NS - \overline{R}$ superstring

$$\alpha' M^2 = 2\left(N^X + N_{NS}^{\psi} + \overline{N}^X + \overline{N}_R^{\psi} - \frac{D-2}{16}\right), \tag{1.114}$$

$$N^X + N_{NS}^{\psi} = \overline{N}^X + \overline{N}_R^{\psi} + \frac{D-2}{16}. \tag{1.115}$$

To obtain the critical dimension, one can consider the $NS - \overline{NS}$ mass shell condition. The lowest value for N^X, $\overline{N}^X$ is 0, while the lowest (non-tachyonic) value for N^ψ_{NS}, $\overline{N}^\psi_{NS}$ is $\frac{1}{2}$. Placing these values into the $NS - \overline{NS}$ mass shell condition results in

$$M^2 = \frac{1}{\alpha'}\left(1 - \frac{D-2}{8}\right). \tag{1.116}$$

The only value of D for which the ground state is massless is

$$D_c = 10, \tag{1.117}$$

thus establishing that superstrings must exist in ten space-time dimensions. The ordering parameters δ, $\overline{\delta}$ in equations (1.90, 1.91), are then

$$\delta_R = 0, \tag{1.118}$$

$$\overline{\delta}_{NS} = \frac{1}{2}. \tag{1.119}$$

There is still a problem regarding the N^ψ_{NS} number operator eigenvalue. Notice that, because the eigenvalue could be 0 in addition to the half integers, this implies

$$\alpha' M^2 = -1 \tag{1.120}$$

which is a tachyon state. Removing this state from the spectrum is vital to a consistent theory.[2] To proceed, one must introduce a projection operator which projects out states according to the following criteria:

- Any NS state created from the vacuum with an even number of creation operators $b^i_{-n-\frac{1}{2}}$, $\overline{b}^i_{-n-\frac{1}{2}}$ is projected out.
- No R state can have spinor components of both chiralities, making the surviving states Majorana-Weyl spinors.

The projection operator is referred to as the GSO (Gliozzi, Scherk, Olive) projection, and must be present in any superstring theory for physical consistency.

[2] There are also other inconsistencies, such as an inexact space-time supersymmetry, which must be addressed. However all such inconsistencies with the formalism are resolved in the same fashion.

1.1.3 Heterotic Strings

The above formalisms describe the consistent massless physical states for a closed superstring. However, absent from the theory is a means of inserting gauge content into closed strings in ten dimensions. One way in which this is done is by creating what is called a heterotic string. The heterotic string is an oriented closed string in which all of the left moving modes are superstring modes in ten space-time dimensions and all of the right moving modes are bosonic string modes in twenty six space-time dimensions. The additional sixteen dimensions for the right moving modes are compactified through periodic identification and form a lattice in which gauge charges are placed.

Writing the action for a heterotic theory can be done in two ways. Firstly, the additional sixteen boson fields can remain as bosons, leaving the theory with two different expressions for space-time. The other, more common way, is to express the additional bosonic degrees of freedom as Majorana-Weyl fermions $\overline{\lambda}^A$, where A indexes the degrees of freedom the fermion modes have. The action (in the conformal gauge) of the fermion based heterotic construction is

$$S = \frac{1}{4\pi\alpha'} \int d^2\xi \left(\partial_a X^\mu \partial^a X_\mu + 2i\psi^\mu \partial_+ \psi_{-,\mu} + 2i \sum_{A=1}^{32} \overline{\lambda}^A \partial_- \overline{\lambda}^A \right). \tag{1.121}$$

While it is possible to quantize this action as was done before, placing additional constraints for quantum mechanical consistency on the compactification of the additional sixteen dimensions yields simplified methods of extracting gauge related information. Therefore discussion of the possible states and gauge charges will be deferred to the next section.

1.2 Weakly Coupled Free Fermionic Heterotic Strings

While the above formalism is useful for deriving some basic information about a theory and its mathematical consistency, phenomenology requires further development. Specifically, the goal of string phenomenology is to extract the massless

spectrum of vacuum states at the string scale and examine their properties. This study focuses on a particular method dubbed the "Weakly Coupled Free Fermionic Heterotic String". This method was pioneered simultaneously by two groups, ABK [6, 7] and KLT [8]. The bosonic modes in the heterotic action (1.121) are re-expressed as non-interacting fermion modes. In this method, toroidal compactifications are initially assumed (though they can be later deformed through specific types of boundary conditions) with radii at the self-dual radius ($R = 1$ in Planck units), which enhances the symmetries of the left moving fermions and the right moving fermions corresponding to compact directions. The massless vacuum spectrum is ultimately determined by choosing the phases the fermion modes gain when transported around non-contractible loops of space-time.

Before making any assumptions about fermion coupling and the compactification radius, the ten boson modes for the left mover as well as the corresponding ten boson modes for the right mover can be expressed as fermions as follows:

$$\partial_a X^\mu \partial^a X_\mu \rightarrow i\psi^{*\mu}\partial_a\psi_\mu + i\overline{\psi}^{*\mu}\partial_a\overline{\psi}_\mu - h\psi^*\psi\overline{\psi}^*\overline{\psi}, \tag{1.122}$$

where h is called the Thirring coupling. The Thirring coupling is a function of the radius of compactification, R, and is taken to zero for the self-dual radius [9]. That is, ψ and $\overline{\psi}$ are free fermions. To directly express the fields (ψ, $\overline{\psi}$) in terms of (X, $\overline{X}$), one introduces the Mandelstam operators

$$\psi = :e^{-iaX}:, \tag{1.123}$$

$$\overline{\psi} = :e^{ia\overline{X}}:, \tag{1.124}$$

where the normal ordering is with respect to the X, $\overline{X}$ creation and annihilation operators (except for the zero modes).

Furthermore, transformations for the boson modes have a different interpretation when they are converted to fermions. As bosons, the fields can undergo

translations (that conserve momentum). Looking at the zero modes,

$$X_0 = \frac{1}{2}(\chi_0 + \tilde{\chi}_0), \tag{1.125}$$

$$\overline{X}_0 = \frac{1}{2}(\chi_0 - \tilde{\chi}_0), \tag{1.126}$$

where χ_0 corresponds to a center of mass translation and $\tilde{\chi}_0$ corresponds to a translation that has no obvious physical meaning, but nonetheless conserves the momentum. It can be shown that shifts of χ_0 are quantized on a circle of radius R, while shifts of $\tilde{\chi}_0$ are quantized on a circle of radius $1/2R$. Shifting χ_0 by $2\pi R$, where R is the compactification radius (infinity for large space-time dimensions), yields the following general transformations[3]

$$\psi \to \psi e^{-i2\pi R/z}, \tag{1.127}$$

$$\overline{\psi} \to \overline{\psi} e^{+i2\pi R/z}. \tag{1.128}$$

Shifting $\tilde{\chi}_0$ by π/R yields the following general transformations

$$\psi \to \psi e^{-i\pi z/R}, \tag{1.129}$$

$$\overline{\psi} \to \overline{\psi} e^{-i\pi z/R}, \tag{1.130}$$

where z is dependent on the Thirring coupling h as follows

$$h = \frac{1}{2}\left(z - \frac{1}{z}\right). \tag{1.131}$$

As previously mentioned, free fermions have $h = 0$, which corresponds to $z = 1$. This reduces the fermion mode transformations to phase shifts dependent on the radius R:

[3] A general heterotic string also allows coupling between the left and right moving boson modes. This causes the left moving fermion phases to be dependent on the gauge content of the theory. This coupling is not considered here.

$$\psi \to \psi e^{-i2\pi R}, \tag{1.132}$$

$$\overline{\psi} \to \overline{\psi} e^{+i2\pi R}, \tag{1.133}$$

$$\psi \to \psi e^{-i\pi/R}, \tag{1.134}$$

$$\overline{\psi} \to \overline{\psi} e^{-i\pi/R}. \tag{1.135}$$

Additionally, setting $R = 1$ or $R = \frac{1}{2}$ for each compactification yields the same transformation rules. This is a manifestation of T-duality, where $R \to 1/R$ leaves the action invariant. It will be shown that this property reduces the number of phases needed to specify a model.

For heterotic strings, the phase transformations of the additional sixteen boson modes coming from the bosonic string must also be examined. The boson modes must span a self-dual even lattice to preserve modular invariance. Thus, they must transform in the following manner

$$\overline{X} \to \overline{X} + 2\pi\alpha_k, \tag{1.136}$$

where α_k is the k^{th} root of a gauge group that makes up the model. This means that the fermionized versions of these modes will transform according to the phase shift

$$\overline{\psi} \to \overline{\psi} e^{-2\pi i\alpha_k}. \tag{1.137}$$

It is now apparent that specifying which phase shifts leave these sixteen boson modes with the correct transformation properties defines the gauge group under which they transform. The phase shifts of the remaining modes determine other properties such as global world sheet charge, supercharge, and spin.

The allowable phases the fermion modes can gain is constrained by world sheet supersymmetry and modular invariance. The supercharge for the world sheet supersymmetry is given by

$$J = \psi^\mu \partial X_\mu + f_{IJK} x^I x^J x^K \tag{1.138}$$

where f_{IJK} are the structure constants of a semi-simple Lie group with a number of generators dependent upon the number of large space-time dimensions and $x^{I,J,K}$ are real compact fermion modes. The allowable choices for the Lie group defined by the structure constants f_{IJK} are very constrained. In particular, the dimension of the group must be $3(10-D)$, where D is the number of large ST dimensions [10]. It is common to choose the simplest Lie group for the world-sheet supersymmetry, $SU(2)^{10-D}$, and that will be the group for the remainder of this study. Other possibilities for $D = 4$ are $SU(3) \otimes SO(5)$ and $SU(2) \otimes SU(4)$, all other even dimensions are products of $SU(2)$. This limits the phase values available to left moving modes. Specifically, these phases can be only either 0 or 1.

Additionally, the action requires that real fermions be placed into complex pairs. However, fermion modes corresponding to compactified superstring dimensions common to the left and right movers may be paired. This pairing is referred to as "rank-cutting," as it removes a $U(1)$ charge from the gauge lattice thereby reducing the total gauge group rank.

The phases gained by all fermion modes when parallel transported around non-contractible loops of space-time define the model. However, modular invariance constrains the allowable values these phases can take (in addition to the constraints placed on the left movers by world-sheet supersymmetry). To derive these constraints, the phases are placed in a vector. For toroidal compactifications, two phase vectors $\vec{\alpha}$, $\vec{\beta}$ are needed to specify the spin(phase)-structure for each of the two non-contractible loops of the torus, though this will eventually be reduced to one phase vector per torus. It will be shown that modular invariance requires the sets of phase vectors $\{\vec{\alpha}\}$, $\{\vec{\beta}\}$ be equal, implying that only one set be chosen. The following discussion will be restricted to the case of rational phases within the range $-1 < \alpha_i \leq 1$. Though some work has been done for deriving the construction rules

of non-rational phases, not all of the consistency requirements for that case have been derived.

Modular invariance is first placed on the vacuum-to-vacuum amplitude. This amplitude (including only the spin-structure dependent terms) is

$$Z_g = \int_{M_g} [D\Omega] \sum_{spin\ structures} c \begin{bmatrix} \vec{\alpha}_1 & \vec{\alpha}_2 & ... & \vec{\alpha}_g \\ \vec{\beta}_1 & \vec{\beta}_2 & ... & \vec{\beta}_g \end{bmatrix} \times \prod_{real\ f} \Theta^{1/2} \begin{bmatrix} \alpha_1(f)...\alpha_g(f) \\ \beta_1(f)...\beta_g(f) \end{bmatrix} (0,\Omega) \times \prod_{complex\ f} \Theta \begin{bmatrix} \alpha_1(f)...\alpha_g(f) \\ \beta_1(f)...\beta_g(f) \end{bmatrix} (0,\Omega), \tag{1.139}$$

where $c \begin{bmatrix} \vec{\alpha}_1 & \vec{\alpha}_2 & ... & \vec{\alpha}_g \\ \vec{\beta}_1 & \vec{\beta}_2 & ... & \vec{\beta}_g \end{bmatrix}$ are the spin-structure coefficients to be constrained by modular invariance momentarily, Ω is the period matrix of the surface encoding the geometry of the g-torus. The summation over the spin-structures accounts for all possible ways to weigh the different phase vectors $\vec{\alpha}_i,\ \vec{\beta}_i$. The products are indexed by the individual fermion modes in each phase vector. $\Theta \begin{bmatrix} \epsilon_1...\epsilon_g \\ \epsilon'_1...\epsilon'_g \end{bmatrix} (0,\Omega)$ is the Jacobi theta function defined by

$$\Theta \begin{bmatrix} \epsilon_1...\epsilon_g \\ \epsilon'_1...\epsilon'_g \end{bmatrix} (z,\Omega) = \sum_{\vec{n}\in Z^g} \exp\left\{ i\pi \left(\vec{n}+\frac{1}{2}\vec{\epsilon}\right) \Omega \left(\vec{n}+\frac{1}{2}\vec{\epsilon}\right) + 2i\pi \left(\vec{n}+\frac{1}{2}\vec{\epsilon}\right)\cdot\left(\vec{z}+\frac{1}{2}\vec{\epsilon}\,'\right) - \frac{i\pi}{2}\vec{\epsilon}\cdot\vec{\epsilon}\,' \right\}, \tag{1.140}$$

where $\vec{\epsilon},\ \vec{\epsilon}\,'$ are the vectors of the individual numbers $\epsilon_i,\ \epsilon'_i$, Ω is the same period matrix in (1.139), and $\vec{n}$ is a vector denoting the possible energy configurations for the Z^g torus. The theta functions essentially evaluate the energy contributions from the towers of possible states for the fermion mode f, sometimes expressed as Verma modules.[4] To summarize, the integration accounts for the degrees of freedom available to the geometry of the torus, the summation of the spin-structures

[4] Verma modules are stacks of weights for a semisimple Lie algebra. The groups represented by each Verma module depend on the gauge content of the model.

correspond to weighing the contributions of the phases, and the theta functions account for the energy contributions of the Verma modules for each fermion mode f. Additionally, the spin-structure coefficients must also factorize in certain limits of the worldsheet geometry. That is,

$$c\begin{bmatrix} \vec{\alpha}_1 & \vec{\alpha}_2 & ... & \vec{\alpha}_g \\ \vec{\beta}_1 & \vec{\beta}_2 & ... & \vec{\beta}_g \end{bmatrix} = c\begin{bmatrix} \vec{\alpha}_1 \\ \vec{\beta}_1 \end{bmatrix} \times c\begin{bmatrix} \vec{\alpha}_2 \\ \vec{\beta}_2 \end{bmatrix} \times \ ... \ \times c\begin{bmatrix} \vec{\alpha}_g \\ \vec{\beta}_g \end{bmatrix}, \tag{1.141}$$

which implies that all spin-structure coefficients can be expressed in terms of one-loop spin-structure coefficients. The spin-structure coefficients can be constrained by demanding equation (1.139) be invariant under the following transformations.

$$\Omega \to \Omega + 1, \tag{1.142}$$

$$\Omega \to \frac{1}{\Omega}, \tag{1.143}$$

$$\Omega \to \Omega - \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \tag{1.144}$$

The first two conditions are transformations that leave the 1-torus topologically invariant. The third is a non-trivial transformation of a genus 2 torus. Forcing the vacuum-to-vacuum amplitude to be invariant under these transformations imposes the following constraints on the spin-structure coefficients.

$$c\begin{bmatrix} \vec{\alpha} \\ \vec{\beta} \end{bmatrix} = \exp\left\{i\pi(\vec{\alpha}\cdot\vec{\alpha} + \vec{\mathbb{1}}\cdot\vec{\mathbb{1}})/4\right\} c\begin{bmatrix} \vec{\alpha} \\ \vec{\beta} - \vec{\alpha} + \vec{\mathbb{1}} \end{bmatrix} \tag{1.145}$$

$$c\begin{bmatrix} \vec{\alpha} \\ \vec{\beta} \end{bmatrix} = \exp\left\{i\pi(\vec{\alpha}\cdot\vec{\beta})/2\right\} c\begin{bmatrix} \vec{\beta} \\ -\vec{\alpha} \end{bmatrix} \tag{1.146}$$

$$\begin{aligned} c\begin{bmatrix} \vec{\alpha} \\ \vec{\beta} \end{bmatrix} \times c\begin{bmatrix} \vec{\alpha}' \\ \vec{\beta}' \end{bmatrix} &= \delta_{\vec{\alpha}}\delta_{\vec{\alpha}'} \exp\left\{-i\pi(\vec{\alpha}\cdot\vec{\alpha}')/2\right\} \\ & c\begin{bmatrix} \vec{\alpha} \\ \vec{\beta} + \vec{\alpha}' \end{bmatrix} \times c\begin{bmatrix} \vec{\alpha}' \\ \vec{\beta}' + \vec{\alpha} \end{bmatrix} \end{aligned} \tag{1.147}$$

where $\delta_{\vec{\alpha}}$ is ± 1, depending on the values of the space-time fermion modes corresponding to large space-time dimensions, $\vec{\mathbb{1}}$ is a vector of all periodic phases, and the dot products are Lorentz dot products in which the dot product of the right moving modes is subtracted from the dot product of the left moving modes. Additionally, real fermion modes are weighed half compared to the complex fermion modes. Equations (1.146, 1.147) can be combined to show

$$c\begin{bmatrix}\vec{\alpha}\\ \vec{\beta}\end{bmatrix}\times c\begin{bmatrix}\vec{\alpha}\\ \vec{\gamma}\end{bmatrix}=\delta_{\vec{\alpha}}\delta_{\vec{\gamma}}c\begin{bmatrix}\vec{\alpha}\\ \vec{\beta}+\vec{\gamma}\end{bmatrix}\times c\begin{bmatrix}\vec{\gamma}\\ \vec{0}\end{bmatrix}. \tag{1.148}$$

This implies that if $\vec{\beta}=\vec{\gamma}=\vec{0}$, then

$$c\begin{bmatrix}\vec{\alpha}\\ \vec{0}\end{bmatrix}=\begin{cases}0\\ \delta_{\vec{\alpha}}c\begin{bmatrix}\vec{0}\\ \vec{0}\end{bmatrix}\end{cases}, \tag{1.149}$$

where $c\begin{bmatrix}\vec{0}\\ \vec{0}\end{bmatrix}$ can be normalized to one. Grouping the set of $\vec{\alpha}$'s for which the spin-structure is nonzero,

$$\Xi=\left\{\vec{\alpha}|c\begin{bmatrix}\vec{\alpha}\\ 0\end{bmatrix}\right\}, \tag{1.150}$$

it can be shown that the elements of Ξ form an additive group. Moreover, it can also be shown that for $c\begin{bmatrix}\vec{\alpha}\\ \vec{\beta}\end{bmatrix}$ to be nonzero, $\vec{\alpha},\vec{\beta}\in\Xi$. This implies that the phase vectors $\vec{\alpha}$ and $\vec{\beta}$ need not be generated in pairs; specifying all of the $\vec{\alpha}$'s for a string vacuum will also include the $\vec{\beta}$'s. Moreover, since Ξ form an additive group, it can be decomposed into a direct sum of integer factors. Therefore, there exists a basis $\{\vec{\alpha}_1^B,...,\vec{\alpha}_k^B\}$ that generates Ξ such that

$$m_1\vec{\alpha}_1^B+...+m_k\vec{\alpha}_k^B=0 \pmod 2 \tag{1.151}$$

The construction restricts itself to bases with $\vec{\alpha}_1^B=\vec{\mathbb{1}}$ as an element of Ξ. In general this element can vary and has its own constraints, but the models constructed herein

(and nearly all WCFFHS models constructed to date) will use $\vec{\mathbb{1}}$. The constraints on the spin-structure (1.145, 1.146, 1.147) can be rewritten as constraints on the basis vectors $\vec{\alpha}_i^B$. Proof of these constraints is tedious and has already been well documented elsewhere. Here they will simply be presented.

Theorem 1.1.

(1) Each model must have the all-periodic vector $\vec{\mathbb{1}}$.

(2) $N_{ij}\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B = 0 \pmod 4$

(3) $N_i\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B = 0 \pmod 8$ *(for* N_i *even)*

(4) The number of simultaneously periodic boundary conditions for any three $\vec{\alpha}^B$*'s is even.*

where N_{ij} is the least common multiple of N_i, N_j. Modular invariance also constrains the allowable values of the spin-structure coefficients. Those are given as follows.

Theorem 1.2.

(1) $$c\begin{bmatrix}\vec{\alpha}_i^B \\ \vec{\alpha}_i^B\end{bmatrix} = \exp\left\{i\pi\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B + \vec{\mathbb{1}}\cdot\vec{\mathbb{1}}/4\right\} c\begin{bmatrix}\vec{\alpha}_i^B \\ \vec{\mathbb{1}}\end{bmatrix}^{N_i/2},$$

(2) $$c\begin{bmatrix}\vec{\alpha}_i^B \\ \vec{\alpha}_j^B\end{bmatrix} = \exp\left\{i\pi\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B/2\right\} c\begin{bmatrix}\vec{\alpha}_j^B \\ \vec{\alpha}_i^B\end{bmatrix}^*.$$

Additionally, constraints are needed to ensure that world-sheet supersymmetry is realized amongst the left movers. To that end, a final condition is needed.

Theorem 1.3.

The boundary conditions for the left movers must obey the Lie algebra for the world-sheet supercharges.

Together, theorems (1.1, 1.2, 1.3) make up the rules for the allowable boundary conditions that can make up a consistent WCFFHS vacuum. To build the model, sectors must be constructed from linear combinations of the basis of $\vec{\alpha}^B$'s. Each sector produces its own set of massless states. The Hilbert space of all possible states from massless sectors does not have modular invariance, however. Modular invariance is derived from the GSO projection, which keeps only a combination of states which leave the vacuum-to-vacuum amplitude invariant under the modular transformations (1.142, 1.143, 1.144).

The states are constructed by applying integral oscillator frequencies to the vacuum fermion mode phases. All such combinations of oscillator frequencies that create zero mass states are checked against the GSO projection. For the weakly coupled free fermionic heterotic string, the oscillator frequencies contribute to the number operator N in the mass shell conditions. Additionally, the fermion modes themselves, even in the vacuum state, also contribute to the mass squared, since in the bosonic language they are momenta. The oscillator frequencies and the phases are combined to form a state vector $\vec{Q}$, defined by $U(1)$ charges. It will be explicitly stated shortly. The mass shell conditions for a free fermionic heterotic string are

$$M_L^2 = \frac{\vec{Q}_L^2}{2} - \frac{1}{2} \tag{1.152}$$

$$M_R^2 = \frac{\vec{Q}_R^2}{2} - 1 \tag{1.153}$$

where $\vec{Q}_L$, $\vec{Q}_R$ are the left and right moving parts of the states, respectively. The states themselves are created by adding oscillator frequencies (similar to a raising/lowering operator) to the modes. The modes contribute one-half their phase value to the mass squared, so the phase values of the sector are divided by two.

$$\vec{Q} = \frac{1}{2}\vec{\alpha} + \vec{F} \tag{1.154}$$

where the $\vec{\alpha}$ here is not necessarily a member of the basis set, but is a linear combination of members of the basis set. $\vec{F}$ can only be combinations of ± 1 and 0, since it

corresponds to anticommuting fermionic raising and lowering operators. Raising by more than that will either cause the state to be massive or prevent it from coupling to the gauge content.

As mentioned previously, the initial set of possible states $\{\vec{Q}_i\}$ do not have modular invariance. To enforce this, the GSO projections must be applied to all of the states. They depend on the spin-structure coefficients and the basis vectors.

$$\exp\left\{i\pi\vec{Q}\cdot\vec{\alpha}_i^B\right\} - \delta_{\vec{\alpha}}c\begin{bmatrix}\vec{\alpha}\\ \vec{\alpha}_i^B\end{bmatrix} = 0 \pmod 2 \tag{1.155}$$

where $\vec{\alpha}$ is the sector that produced the state. The above equation must hold for all members of the basis set $\vec{\alpha}_i^B$. Note that there is a choice in the spin-structure coefficients which affects the states that pass the GSO projections. This means identical basis vectors can, with different spin-structure choices, result in different models. Additionally, using the $c\begin{bmatrix}\vec{\alpha}_i^B\\ \vec{\alpha}_j^B\end{bmatrix}$ constraints on the spin-structure are computationally cumbersome. This can be remedied by transforming them into rational values. This is done by rewriting $c\begin{bmatrix}\vec{\alpha}_i^B\\ \vec{\alpha}_j^B\end{bmatrix}$ as

$$c\begin{bmatrix}\vec{\alpha}_i^B\\ \vec{\alpha}_j^B\end{bmatrix} = (-1)^{s_i+s_j}\exp\left\{i\pi k_{ij}\right\}\exp\left\{-i\pi\vec{\alpha}_i^B\cdot\vec{\alpha}_j^B\right\}, \tag{1.156}$$

where $s_{i(j)}$ is 0 if $\vec{\alpha}_{i(j)}^B$ is a space-time boson (antiperiodic space-time modes ψ^μ) and 1 if it is a space-time fermion (periodic space-time modes ψ^μ). The modular invariance constraints on the spin-structure can now be rewritten as constraints on k_{ij}.

$$N_j k_{ij} = 0 \pmod 2, \tag{1.157}$$

$$k_{ij} + k_{ji} = \frac{1}{2}\vec{\alpha}_i^B\cdot\vec{\alpha}_j^B \pmod 2, \tag{1.158}$$

$$k_{ii} + k_{i1} = \frac{1}{4}\vec{\alpha}_i^B\cdot\vec{\alpha}_i^B - s_i \pmod 2, \tag{1.159}$$

where s_i is defined as above, and N_j is the order of the j^{th} $\vec{\alpha}^B$ in the basis set of phases. The GSOPs can also be rewritten in terms of k_{ij},

$$\vec{\alpha}_j^B \cdot \vec{Q}_{\vec{\alpha}} = \sum_i k_{ji} a_i + s_j \qquad (\text{mod } 2) \tag{1.160}$$

where s_i is defined again as above, and a_i are the coefficients of $\vec{\alpha}^B$'s for the sector $\vec{\alpha}$ that produced the state $\vec{Q}_{\vec{\alpha}}$. The two groups who created the WCFFHS formalism, ABK and KLT, use $c \begin{bmatrix} \vec{\alpha} \\ \vec{\beta} \end{bmatrix}$ and k_{ij}, respectively. The studies herein will use the more computationally advantageous k_{ij} formalism.

1.3 The String Vacuum Landscape

Thus far, three general types of string theories have been discussed: the bosonic string, the superstring, and the heterotic string. However, there are theoretical details that outline different formalisms amongst the superstrings and heterotic strings. For the non-heterotic superstring theories, the main differences lie in the "orientation" — a quantum mechanical quantity that distinguishes two otherwise identical strings — the chirality of the left and right movers, and the number of world-sheet supercharges that generate supersymmetry. While the details are numerous, the basic differences amongst the non-heterotic theories can be laid out as follows.

The superstring theory that has open and closed non-oriented strings and 16 WS supercharges (giving $N = 1$ WS SUSY) is called the Type I superstring theory. Two other superstring theories exist, both with 32 WS supercharges which generate $N = 2$ WS SUSY. They are both theories of oriented strings, and the formal difference between the two is that one is non-chiral, while the other is chiral. They are called Type IIA and Type IIB, respectively.

There are two formally different heterotic string theories as well, the difference between them being the gauge group of the lattice making up additional sixteen

dimensions the bosonic right mover has. They are denoted $SO(32)$ and $E_8 \otimes E_8$ heterotic theories.[5]

For a time, these five string theories presented a dilemma: if there is more than one way to solve a problem, what makes one solution different than another if all of them are physically and mathematically consistent? That was answered when it was shown that by examining certain dualities and adding an additional dimension (from 10 to 11), all of the five string theories are actually low energy projections of a larger theory, called M-theory. There are two dualities that relate the theories amongst each other. S-duality inverts the coupling strength of the string, meaning that one theory's strong coupling limit is its S-dual's weak coupling limit. This is useful because weak coupling allows perturbative methods to be used to make phenomenological calculations. The other duality inverts the radius (in Planck scale units), so that a theory compactified with radius R will have a T-dual theory with compactified radius $1/R$. Combined with low energy projections, these dualities are all that is needed to combine the five superstring theories into one large theory. These dualities are summarized in figure 1.1. While this is the (nearly) complete string theory picture in ten dimensions, compactifying six of the space-time dimensions to obtain the necessary four that describe the physical universe leads to a multitude of different configurations of gauge groups, matter states, space-time supersymmetries, and cosmological constants, all of which are consistent with the rules of string theory. It was for a time thought that on the order of a hundred trillion geometric configurations (called Calabi-Yau manifolds) were consistent.

In an attempt to show how the proper cosmological constant could be explained from string theory, it was discovered that the compactified dimensions can have field fluxes through them and still remain stable. These fluxes are quantized,

[5] The WCFFHS method of constructing string models does not distinguish between which heterotic theory is being used, since the compactifications are at the self-dual radius. Rather, these two gauge groups appear, along with a few other models with different gauge groups, by choosing certain phases for the right mover. More details will be presented regarding this connection later.

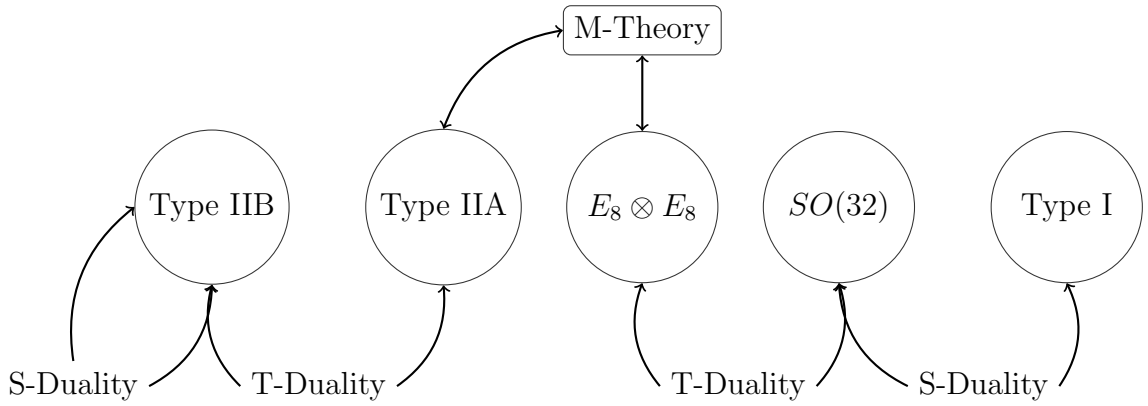


Figure 1.1: The five consistent ten-dimensional string theories and the dualities that relate them.

but very numerous, and have been described as forming a "discrete continuum" or "discretum"[11]. Later work revealed that though the number of stable geometries with fluxes is finite, the number of configurations (and thus the number of string vacua) is extremely high. Rough estimates put this number at approximately 10^{500} different configurations per Calabi-Yau manifold [12]. The vast number of stable, consistent string vacua has come to be known as the "string vacuum landscape."

The dualities between the five classes of string theories has led to a concerted effort in locating patterns amongst large numbers of explicitly constructed string models for all construction methods. Doing this has required a large scale effort involving multiple disciplines including not only physics and mathematics but also computer science. Two organizations, each called the "String Vacuum Project" have formed in the United States and Europe as a way of centralizing the research needed to explore the string theory landscape.

Work has already been done in exploring possible WCFFHS vacua statistically [13, 14, 15, 16]. However, these searches have been random. Random searches include several difficulties involving correlations that change as a function of sample size [17]. Additionally, though the construction has been worked out for any rational phase, the large scale searches to date have basis vectors with only periodic/antiperiodic

modes. Presented here is the first in a series of investigations in which the basis vectors and GSO coefficient matrices are investigated completely and systematically. Specialized software optimized for speed, generality, and ease of use has been built from scratch for this purpose.

The ultimate goal of examining string vacua is to find the proper geometry that describes the universe. Statistical studies are used to draw correlations between the geometric inputs and particle spectra. These correlations will aid in finding phenomenologically viable "patches" of the landscape, and hopefully to a complete string-derived description of the universe.

CHAPTER TWO

Constructing Free Fermionic Heterotic String Models

2.1 Introduction

WCFFHS models have produced some of the most phenomenologically realistic effective field theories resulting from a string model [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53] . The ease at which these models can be generated by a computer enhances the appeal. This document outlines the steps needed to create WCFFHS models without explicitly referencing any one piece of software. The discussion here will be limited to models with four large space-time dimensions, though the process can be easily generalized to models with fewer large space-time dimensions.

2.2 Inputs

This section details the elements needed to specify a WCFFHS model. There are two inputs which are needed: a set of 64 component basis vectors and an $L \times L$ matrix (where L is the number of basis vectors), called the k_{ij} matrix. The basis vectors represent phases that real fermion modes gain when parallel transported around non-contractible loops of space-time. These essentially specify the "geometry" of the model. That is, the phases occur due to boundary conditions on the world sheet, so when the phases are specified, the boundary conditions are also specified. Sometimes these vectors are called boundary vectors for this reason. The fact that these basis vectors have 64 elements is a result of the number of space-time compactifications, six in this case. Different numbers of compactified directions will result in different basis vector sizes. The k_{ij} matrix is a matrix of coefficients used in the GSO (Gliozzi, Scherk, Olive) projection equations. These are sometimes re-

ferred to as GSO coefficients, and there are other equivalent ways to express these coefficients besides the k_{ij} matrix.

2.2.1 Anatomy of a Basis Vector

There's a little bit of terminology needed before proceeding forward. The order specifies the range of values the basis vector can take, starting with zero. So an order N basis vector can have elements up to N-1. Since these are phases, the physical values (in units of π) are fractional. However, fractions are not ideal for computer coding, so they are often coded into integers in the following way:

$$\alpha_i^B = \frac{2m_i}{N}, -1 < \alpha_i^B \leq 1, \tag{2.1}$$

where α_i^B is the physical value of the phase which has been adjusted to fit the limits described. m_i is the integer code which is typically used to describe the vector in computer programs. The range of m_i is 0 to $N-1$.

A model can have any number of basis vectors, and each basis vector making up the model is also referred to in generic discussions as a layer.

A basis vector in four large ST dimensions is made up of 64 components, but different components in the vector mean different things and have different rules and constraints. The first 20 elements are referred to as the left moving part, and they represent the supersymmetric side of the heterotic string. Supersymmetry (SUSY) here refers to world-sheet (WS) supersymmetry. Space-time supersymmetry is determined by the basis vectors and the GSO coefficients. The number of large ST dimensions in the theory determines the number of real ST fermions ($\psi^{1,\cdots}$). The number of compactified directions specifies the number of real fermions and their bosonic superpartners corresponding to those directions. These are written as a triplet, since the fermion ($x^{1,\cdots}$, sometimes written as $\chi^{1,\cdots}$) and the boson field expressed as two fermions ($y^{1,\cdots}, w^{1,\cdots}$, sometimes written as $y^{1,\cdots}, \omega^{1,\cdots}$) all correspond to the same compactified direction. From this point forward, the phases in the basis

alphas will be denoted by the name of the field acquiring the phase. Thus in this notation the statement $\psi^1 = 1$ means the field ψ^1 gains a phase equal to one. This makes the discussion of basis alpha elements less cumbersome.

The equations which give the number of $\psi^{1,\cdots}, x^{1,\cdots}$and $y^{1,\cdots}, w^{1,\cdots}$ (in light-cone gauge) are as follows:

$$\text{Number of Real ST Fermions, } \psi^{1,\cdots} = D - 2, \tag{2.2}$$

$$\text{Number of Real Fermion Triplets, } (x, y, w)^{1,\cdots} = 10 - D. \tag{2.3}$$

Thus, for $D = 4$, there are two space-time fermions, and six real fermion triplets. Schematically, then, the left mover looks like:

$$(\psi^1\ \psi_c^1), (x\ y\ w)^1, (x\ y\ w)^2, (x\ y\ w)^3, (x\ y\ w)^4, (x\ y\ w)^5, (x\ y\ w)^6, \tag{2.4}$$

where the superscript on the triplets is an index. Note that the $(\psi^1\ \psi_c^1)$ are labeled with the same index. The reason for this is that the two real space-time fermions form a complex pair, meaning:

$$\psi^1 = \psi_c^1 \tag{2.5}$$

in every basis vector. These two could be, and often are, expressed as a single element in the vector. In this document they are expressed as two real, but always equal, fermion modes. Physically these elements, when present in a state (discussed later), determine whether that state will be a space-time boson ($\psi^1 = \psi^{1*} = 0$) or a space-time fermion ($\psi^1 = \psi^{1*} = 1$). These elements also play a role in equations for the GSO coefficient matrix and the GSO projection, which will also be discussed later. In addition, there are rules that must also be followed for the $(x\ y\ w)$ triplets. The first rule is that the x^i form complex pairs also,

$$x^1 = x^2, \tag{2.6}$$

$$x^3 = x^4, \tag{2.7}$$

$$x^5 = x^6, \tag{2.8}$$

Again, these could have been written as three fermions to reduce the elements, but this loses the geometric significance of the schematic (2.4). As mentioned in the first chapter, the WS SUSY can be a representation of one of several Lie algebras [10]. However, not all of these can yield models with $N = 1$ ST SUSY [54]. Most models use the simplest, $SU(2)^6$, the superscript in this case indicating the number of tensor products. A consequence of selecting this SUSY generator is that the x values must be paired as in equations (2.6, 2.7, 2.8). Another consequence of this selection is that the left movers are either periodic ($\alpha_i^B = 1$) or anti-periodic ($\alpha_i^B = 0$). That is, the left mover is order 2. Other choices for the WS supercurrent's symmetry may result in higher order LM modes.

The other rule to follow is one of spin. The ST fermion modes being compactified into triplets are real fermions. In order to preserve their spin, another rule must be imposed upon the allowable phases of the modes in a compact triplet.

$$\text{The number of periodic modes within a given triplet is odd.} \tag{2.9}$$

The remaining 44 elements[1] are referred to as the right moving part of the basis vector. These describe the bosonic side of the heterotic string, so there is no world sheet SUSY here. Instead, there are more space-time dimensions (since bosonic string theories have 26 space-time dimensions as opposed to 10). The 44 elements are split into three categories for convenience: observable, compactified, hidden. The observable and the hidden elements are mirrors of one another, and each contains 16 of the RM real fermion modes. Here is a schematic of each:

$$\text{Obs: } (\bar{\psi}^1\ \bar{\psi}_c^1\ \bar{\psi}^2\ \bar{\psi}_c^2\ \bar{\psi}^3\ \bar{\psi}_c^3\ \bar{\psi}^4\ \bar{\psi}_c^4\ \bar{\psi}^5\ \bar{\psi}_c^5\ \bar{\eta}^1\ \bar{\eta}_c^1\ \bar{\eta}^2\ \bar{\eta}_c^2\ \bar{\eta}^3\ \bar{\eta}_c^3), \tag{2.10}$$

$$\text{Hid: } (\bar{\phi}^1\ \bar{\phi}_c^1\ \bar{\phi}^2\ \bar{\phi}_c^2\ \bar{\phi}^3\ \bar{\phi}_c^3\ \bar{\phi}^4\ \bar{\phi}_c^4\ \bar{\phi}^5\ \bar{\phi}_c^5\ \bar{\phi}^6\ \bar{\phi}_c^6\ \bar{\phi}^7\ \bar{\phi}_c^7\ \bar{\phi}^8\ \bar{\phi}_c^8). \tag{2.11}$$

[1] There are space-time boson models in WCFFHS models which are used to generate the graviton. Gravity is not the primary concern of the present study. Thus, the right moving space-time boson elements are not included in the basis vectors.

The elements $\bar{\eta}^1, ..., \bar{\eta}^3_c$ are notated as such because they are often (but not always) associated with the x^i, x^i_c values on the LM side. These are, as the notation indicates, complex pairs, so the following is true:

$$\bar{\psi}^i = \bar{\psi}^i_c, \tag{2.12}$$

$$\bar{\eta}^i = \bar{\eta}^i_c, \tag{2.13}$$

$$\bar{\phi}^i = \bar{\phi}^i_c, \tag{2.14}$$

for $i = 1, ..., 8$ for $\bar{\psi}, \bar{\phi}$ and $i = 1, ..., 3$ for $\bar{\eta}$.

Recall that the heterotic string without compactifications has only two possible gauge groups that are consistent with modular invariance: $SO(32)$ and $E_8 \otimes E_8$. There are differences between the two constructions at the string scale, but the low energy effective field theories (for nine and lower large ST dimensions) resulting from the two are indistinguishable. Since that is the primary concern of this type of model building, either construction will serve the same purpose. The reason the RMs are classified here as observable and hidden is due to the split between the 16 real fermion modes that make up each E_8 in the $E_8 \otimes E_8$ model.[2] This is convenient because there are a lot of elements to keep track of, and splitting them according to the $E_8 \otimes E_8$ makes it easier. The terms "observable" and "hidden" are relative as well. When models were constructed individually, the observable sector gauge group was placed with the $\bar{\psi}^{1,...,5}$ elements. Generically, the observable sector gauge group can be generated by any of the right moving elements that form complex pairs. The observable and hidden modes are complex, (equations 2.12, 2.13, 2.14), and thus expressible as two sets of eight. For clarification they are kept in the real basis throughout this chapter.

[2] The matter states charged under the subgroups of each E_8 are often independent. This effectively splits the ten dimensional RM modes into separate groups of sixteen. The notation is an artifact of that independence.

The compactified elements are a result of the six compactifications. Recall that each compactification generates a fermion and a boson (expressed as two real fermions) on the LM side of the vector. The right side gains only the boson, since it is a bosonic string. Therefore, while the number of observable and hidden complex fermions remains at 32 for any compactification, the number of compactified real fermions change according to the following rule:

$$\text{Number of real compactified RM fermions, } \bar{y}^i \ \bar{w}^i = 2(10 - D). \tag{2.15}$$

These compactified real fermions are notated as $\bar{y}^i$ and $\bar{w}^i$ and are schematically written as follows:

$$\text{Compact: } (\bar{y}^1 \ \bar{y}^2 \ \bar{y}^3 \ \bar{y}^4 \ \bar{y}^5 \ \bar{y}^6 \ \bar{w}^1 \ \bar{w}^2 \ \bar{w}^3 \ \bar{w}^4 \ \bar{w}^5 \ \bar{w}^6). \tag{2.16}$$

The notation here indicates two things: firstly, these are the right mover versions of the left moving boson parts of the compactified triplets. Hence they are given the same variable name with a bar over it. In addition, these are not placed into complex pairs by construction. They follow a different rule:

Compactified real RM fermions may be paired with either LM or RM modes. (2.17)

This is followed up by another set of statements.

$$\text{Each real fermion mode can only be paired once.} \tag{2.18}$$

$$\text{All real fermion modes must be paired.}^3 \tag{2.19}$$

This means that there are no fermion modes which are not paired with another fermion. It's obvious that the left moving ST fermions and the x modes follow both of these conditions, as well as the Observable and the Hidden parts of the RM modes. The pairings amongst the $y^i \ w^i \ \bar{y}^i \ \bar{w}^i$ must be consistent with (2.19). In summary, then, the total schematic for a right mover is as follows:

$$(\bar{\psi}^1 \ \bar{\psi}^1_c \ ... \ \bar{\psi}^5 \ \bar{\psi}^5_c \bar{\eta}^1 \bar{\eta}^1_c ... \bar{\eta}^3 \bar{\eta}^3_c \ \bar{y}^1 \ ... \ \bar{y}^6 \ \bar{w}^1 \ ... \ \bar{w}^6 \ \bar{\phi}^1 \ \bar{\phi}^1_c \ ... \ \bar{\phi}^8 \ \bar{\phi}^8_c) \tag{2.20}$$

The pairings of matching fermion modes must match for all basis vectors in a given model. Obviously this is true for the ST fermions, the x's, and the observable and hidden parts. For the $y, w, \bar{y}, \bar{w}$ this acts as a constraint for allowed basis vectors. This is most easily seen with an example. Consider the following elements of a few basis vectors.

	y^1	y^2	w^5	w^6	$\bar{y}^1$	$\bar{y}^2$	$\bar{w}^5$	$\bar{w}^6$
b_1	(1	1	1	1	1	1	1	1)
b_2	(1	0	1	0	1	0	0	1)
b_3	(1	0	0	0	0	1	1	1)

Taking a look at the pairings starting with b_1, the following elements have matching boundary conditions:

$$\{y^1\ y^2\ w^5\ w^6\ ||\ \bar{y}^1\ \bar{y}^2\ \bar{w}^5\ \bar{w}^6\}.$$

By default it is assumed that the adjacent elements are paired, meaning that if b_1 is the only basis vector in the model, the following elements would be paired:

$$\{y^1\ y^2\}\ \{w^5\ w^6\}\ \{\bar{y}^1\ \bar{y}^2\}\ \{\bar{w}^5\ \bar{w}^6\}.$$

When b_2 is added, this splits the set of eight elements as follows:

$$\{y^1\ w^5\ ||\ \bar{y}^1\ \bar{w}^6\}\ \{y^2\ w^6\ ||\ \bar{y}^2\ \bar{w}^5\}.$$

Therefore, if b_1 and b_2 are the only basis vectors in the model, the following elements would be paired:

$$\{y^1\ w^5\}\ \{\bar{y}^1\ \bar{w}^6\}\ \{y^2\ w^6\}\ \{\bar{y}^2\ \bar{w}^5\}.$$

Adding b_3 to the model puts the simultaneous matching boundary conditions into pairs, thus completely determining the pairings:

$$\{y^1\ \bar{w}^6\}\ \{w^5\ \bar{y}^1\}\ \{y^2\ w^6\}\ \{\bar{y}^2\ \bar{w}^5\}.$$

Now suppose another basis vector, b_4, is added.

	y^1	y^2	w^5	w^6	$\bar{y}^1$	$\bar{y}^2$	$\bar{w}^5$	$\bar{w}^6$
b_1	1	1	1	1	1	1	1	1
b_2	1	0	1	0	1	0	0	1
b_3	1	0	0	0	0	1	1	1
b_4	1	1	0	0	0	0	0	0

Only the following can be paired in each basis vector:

$$\{w^5\ \bar{y}^1\}\ \{\bar{y}^2\ \bar{w}^5\}.$$

y^1 and y^2 match for basis vector b_4, but not for b_3. The same is true for w^6 and $\bar{w}^6$. There is no choice for pairing these elements such that all of them are paired for each basis vector. Therefore, this model is inconsistent by (2.19).

There is an ordering redundancy for parts of the basis vector as well. Consider the following two order 2 basis vectors b_1 and b_2.

	$\bar{\phi}^1$	$\bar{\phi}^1_c$	$\bar{\phi}^2$	$\bar{\phi}^2_c$	$\bar{\phi}^3$	$\bar{\phi}^3_c$	$\bar{\phi}^4$	$\bar{\phi}^4_c$	$\bar{\phi}^5$	$\bar{\phi}^5_c$	$\bar{\phi}^6$	$\bar{\phi}^6_c$	$\bar{\phi}^7$	$\bar{\phi}^7_c$	$\bar{\phi}^8$	$\bar{\phi}^8_c$
b_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
b_2	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

This is identical to the following model:

	$\bar{\phi}^1$	$\bar{\phi}^1_c$	$\bar{\phi}^2$	$\bar{\phi}^2_c$	$\bar{\phi}^3$	$\bar{\phi}^3_c$	$\bar{\phi}^4$	$\bar{\phi}^4_c$	$\bar{\phi}^5$	$\bar{\phi}^5_c$	$\bar{\phi}^6$	$\bar{\phi}^6_c$	$\bar{\phi}^7$	$\bar{\phi}^7_c$	$\bar{\phi}^8$	$\bar{\phi}^8_c$
b_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
b_2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

This redundancy can be described with the following rule:

The ordering of a set of complex fermion modes does not affect the phenomenology as long as the other basis vectors have identical boundary conditions for those modes. (2.21)

Therefore, adding a third basis vector b_3 of order 4 yields:

	$\bar{\phi}^1$	$\bar{\phi}^1_c$	$\bar{\phi}^2$	$\bar{\phi}^2_c$	$\bar{\phi}^3$	$\bar{\phi}^3_c$	$\bar{\phi}^4$	$\bar{\phi}^4_c$	$\bar{\phi}^5$	$\bar{\phi}^5_c$	$\bar{\phi}^6$	$\bar{\phi}^6_c$	$\bar{\phi}^7$	$\bar{\phi}^7_c$	$\bar{\phi}^8$	$\bar{\phi}^8_c$
b_1	(1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1)
b_2	(0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1)
b_3	(3	3	2	2	0	0	1	1	1	1	0	0	3	3	2	2)

The model produced by this set of basis vectors is identical to the model produced by the set of basis vectors:

	$\bar{\phi}^1$	$\bar{\phi}^1_c$	$\bar{\phi}^2$	$\bar{\phi}^2_c$	$\bar{\phi}^3$	$\bar{\phi}^3_c$	$\bar{\phi}^4$	$\bar{\phi}^4_c$	$\bar{\phi}^5$	$\bar{\phi}^5_c$	$\bar{\phi}^6$	$\bar{\phi}^6_c$	$\bar{\phi}^7$	$\bar{\phi}^7_c$	$\bar{\phi}^8$	$\bar{\phi}^8_c$
b_1	(1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1)
b_2	(0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1)
b_3	(0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3)

Any other basis vectors added to b_1, b_2, b_3 cannot be reordered because the b_1, b_2, b_3 basis vectors have no matching boundary conditions for the complex fermions.

Basis vectors must have modular invariance, which is essentially a reparametrization invariance on the world sheet which prevents certain anomalies from appearing in the theory. The equations/rules for this are straightforward:

$$N_{ij}\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B = 0 \pmod 8 \tag{2.22}$$

$$N_{ii}\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B = 0 \pmod{16} \text{ (even orders only)} \tag{2.23}$$

$$\text{The number of simultaneously periodic real fermions for any three basis vectors is even.} \tag{2.24}$$

where N_{ij} is the lowest common multiple of the orders of the two basis vectors, N_i and N_j. The modulus operations are chosen as such because the basis vectors are are expressed in a real basis. In addition, odd ordered basis vectors ignore equation (2.23) and instead use only equation (2.22). The third statement, (2.24), refers to matching periodic boundary conditions amongst three basis vectors, and the three basis vectors may contain multiple copies of basis vectors in the set for the model.

This ensures that all fermion modes are able to be paired. Thus, (2.19) is satisfied as long as (2.24) is satisfied.

The dot products in equations (2.22) and (2.23) are referred to as Lorentz dot products, which take the following form.

$$\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B = \vec{\alpha}_i^{B,L} \cdot \vec{\alpha}_j^{B,L} - \vec{\alpha}_i^{B,R} \cdot \vec{\alpha}_j^{B,R}, \tag{2.25}$$

where the dot products on the right hand side of (2.25) are regular dot products. Some write this as *right-left*, which is also consistent. This document always uses the *left-right* convention. There are other "special" dot products used throughout the construction process. These will be noted as they occur.

2.2.2 The k_{ij} Matrix

The k_{ij} matrix represents a degree of freedom that is granted by modular invariance, an essential component needed for a physically consistent WCFFHS model (described in the section 2.2.1). It is an $L \times L$ matrix, where L is the number of basis vectors in the model. Each row and column corresponds to a basis vector in the model. Different choices for the k_{ij} coefficients will result in a different model. These coefficients play into the GSO projection equations, which select the states that can be simultaneously kept in a model. This will be covered in a later section. There is essentially one basic rule for selecting k_{ij} values, and another for ensuring that the k_{ij} matrix follows modular invariance:

$$N_j k_{ij} = 0 \pmod 2. \tag{2.26}$$

In essence, this means that the order of the column of the k_{ij} matches the order of the basis vector. Note that this is the order of the entire basis vector rather than the right moving part. In addition, modular invariance also imposes constraints on

the values of k_{ij} relative to one another. The following two rules apply:

$$\frac{1}{2}\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B = k_{ij} + k_{ji} \ (mod\ 2) \tag{2.27}$$

$$\frac{1}{4}\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B - s_i = k_{ii} + k_{i1} \ (mod\ 2) \tag{2.28}$$

In these equations, $\vec{\alpha}_{i,j}^B$ are the basis vectors associated with the rows and column in question, and s_i is the space-time value of $\vec{\alpha}_i^B$. That is,

$$s_i = \begin{cases} 0 \ (\vec{\alpha}_i^B \text{ is ST boson}) \\ 1 \ (\vec{\alpha}_i^B \text{ is ST fermion}) \end{cases} \tag{2.29}$$

These equations can be solved to constrain most of the k_{ij} matrix. Aside from the k_{11} element, the diagonals and one half of the off-diagonal elements are determined completely by the basis vectors and the other half of the off-diagonal elements. In general, the k_{11} element as well as the lower off diagonal (below the diagonal) elements are specified, and equations (2.27, 2.28) are solved for the other half. An example for a model with five order-two basis vectors and one additional order four basis vector would have an input for the k_{ij} matrix like this:

$$\begin{pmatrix} 1 & \square & \square & \square & \square & \square \\ 0 & \square & \square & \square & \square & \square \\ 1 & 1 & \square & \square & \square & \square \\ 1 & 1 & 1 & \square & \square & \square \\ 1 & 1 & 1 & 1 & \square & \square \\ 1 & 1 & 1 & 1 & 1 & \square \end{pmatrix} \tag{2.30}$$

Here the $\square$ means an element determined by the above equations (2.27, 2.28). Notice the last column is completely determined, and is also order four. The modular invariance equations (2.27, 2.28) for the k_{ij} matrix will ensure that the values for the last column take on the proper values. In principle, any off-diagonal elements

(and any one diagonal element) could have been used as inputs so long as the corresponding element for each chosen in the equations (2.27, 2.28) does not violate modular invariance. Had the upper half of the matrix been chosen, the only consistent solutions would have the lower off-diagonal components be 0 or 1 (mod 2). In (2.30) the lower off-diagonal elements are chosen as for convenience.

2.3 Generating States

This section will explain the mathematical details of obtaining the physical states that a given set of $\vec{\alpha}^B$'s generates. First, the full space must be built out of the $\vec{\alpha}^B$'s. These vectors are labeled simply $\vec{\alpha}$. Next, the states are generated using the raising and lowering operators on the individual modes for each $\vec{\alpha}$. Finally, the GSO projection equations will eliminate states which cannot simultaneously exist in a theory.

2.3.1 $\vec{\alpha}^B$'s

The $\vec{\alpha}^B$'s form the basis of the space in which the states are built. Care must be taken when building this basis, however, as the integer coding needs to be translated to physical values per equation (2.1). For example, suppose there is an order 4 basis vector with the following elements:

$$\vec{v} = (1\ 1)(1\ 0\ 0)^6||(1\ 1\ 1\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 0\ 0\ 0\ 0\ 0\ 0\ 1^6\ 3^6\ 0^{16}). \tag{2.31}$$

To convert this basis vector to its basis alpha form, consolidate the orders of the left and right movers into the least common multiple between them, which is 4,

$$\vec{\alpha}^B = (1\ 1)(1\ 0\ 0)^6||(\frac{1}{2}\ \frac{1}{2}\ \frac{1}{2}\ \frac{1}{2}\ \frac{1}{2}\ \frac{1}{2}\ 1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ \frac{1}{2}^6\ -\frac{1}{2}^6\ 0^{16}). \tag{2.32}$$

The 3's are converted to fit into the range $-N < \alpha_i^B \leq N$ into –2's. For an odd order, such as 3, more care must be taken, since order 3 does not actually have periodic modes. Consider the following:

$$\vec{v} = (1\ 1)(1\ 0\ 0)^6||(1\ 1\ 1\ 1\ 1\ 1\ 2\ 2\ 2\ 2\ 0\ 0\ 1\ 1\ 2\ 2\ 0^6\ 1^6\ 0^{16}). \tag{2.33}$$

First, convert the right moving side to fractions. $1 \to \frac{2}{3}$ and $2 \to -\frac{2}{3}$. Then find the least common multiple of 2 and 3, which is 6. Scale the appropriate fractions up to that order. For the left side, $1 \to \frac{2}{2} \to \frac{6}{6}$. For the right movers, $\frac{2}{3} \to \frac{4}{6}$ and $-\frac{2}{3} \to -\frac{4}{6}$. Thus, this basis vector is converted into the following basis alpha:

$$\vec{\alpha}^B = (1\ 1)(1\ 0\ 0)^6||(\frac{2}{3}\ \frac{2}{3}\ \frac{2}{3}\ \frac{2}{3}\ \frac{2}{3}\ \frac{2}{3}\ -\frac{2}{3}\ -\frac{2}{3}\ -\frac{2}{3}\ -\frac{2}{3}\ 0\ 0\ -\frac{2}{3}\ -\frac{2}{3}\ 0^6\ \frac{2}{3}^6\ 0^{16}). \quad (2.34)$$

2.3.2 $\vec{\alpha}$'s

To form the full space of $\vec{\alpha}$'s from the $\vec{\alpha}^B$'s, take the linear combinations of the $\vec{\alpha}^B$'s.

$$\vec{\alpha} = \sum_{i=1}^{m} a_i \vec{\alpha}_i^B, \quad (2.35)$$

where the index i goes over all of the $\vec{\alpha}_i^B$'s for the model, and a_i is an integer coefficient with a range of values from 0 to $N_i - 1$, where N_i is the full order (denominator) of the $\vec{\alpha}_i^B$. The elements of the $\vec{\alpha}$ must conform to the same value range as the elements of the $\vec{\alpha}^B$, and often need to be converted to fit the proper range (2.1). This formula is applied for each possible value of a_i until the full space is built. It is important when calculating the $\vec{\alpha}$'s that the $\vec{\alpha}^B$'s have a common denominator as well, since they are added as fractions together.

Once built, the $\vec{\alpha}$'s will then be used to generate the physical states of the model. Not all $\vec{\alpha}$'s can produce massless states, however. To see which $\vec{\alpha}$'s can produce massless states for a theory, the left and right moving lengths-squared must be calculated as follows

$$\vec{\alpha}_L \cdot \vec{\alpha}_L \leq 8, \quad (2.36)$$

$$\vec{\alpha}_R \cdot \vec{\alpha}_R \leq 16, \quad (2.37)$$

where the limiting values are in the real basis. In a complex basis, the values are halved. As will be described in section (2.3.3), any $\vec{\alpha}$'s which have lengths-squared

above the limits in (2.36) and (2.37) will not produce massless states, and can be ignored.

The different states in a theory will come to represent different force, matter, and SUSY partner particles. This is the rule for determining which $\vec{\alpha}$'s will produce what types of states.

$$\text{Fermion: LM ST modes are periodic.} \tag{2.38}$$

$$\text{Boson: } \vec{\alpha}_L^2 = 0. \tag{2.39}$$

$$\text{SUSY partner: } \vec{\alpha}_L^2 = 8,\ \vec{\alpha}_R^2 = 0. \tag{2.40}$$

Once the type of $\vec{\alpha}$ is determined, the states can be built. There are many shortcuts to producing the states based on which $\vec{\alpha}$ serves as the generator for the states. These shortcuts are outlined in the next section.

2.3.3 States

To build the physical states for the theory, the following equation is applied:

$$\vec{Q} = \frac{\vec{\alpha}}{2} + \vec{F}, \tag{2.41}$$

where $\vec{Q}$ is the state vector defining the charges and $\vec{F}$ is the raising or lowering operator for a particular fermion mode in the state vector.

$$F_i = \pm 1 \tag{2.42}$$

The physical states of the model are produced by applying the raising/lowering operator all possible ways such that the state is massless. Since the model is constructed at string scale energy, anything massive at the string scale will be far too massive to observe at standard model energy levels. Thus, only the massless states are considered in FFHS models. The masslessness conditions for the left and right movers

with regard to the states are

$$m_L^2 = \frac{1}{4}\vec{Q}_L^2 - \frac{1}{2}, \tag{2.43}$$

$$m_R^2 = \frac{1}{4}\vec{Q}_R^2 - 1. \tag{2.44}$$

This constrains the values of $\vec{Q}^2$ as follows:

$$\vec{Q}_L^2 = 2, \vec{Q}_R^2 = 4. \tag{2.45}$$

The above conditions are what provide the limits in equations (2.36, 2.37). It is also important to note that the $\vec{F}$ operator applies to complex pairs in the same way, since the complex fermion mode pairs must always be equal. The easiest way to show how states are produced is with an example, so consider the following vector of five right moving complex elements.

$$\bar{\psi}^{1,...,5} = (0, ..., 0) \tag{2.46}$$

The possible charge states produced by applying $\vec{F}$ to these elements are shown in Table 2.1.

Table 2.1: The possibe charge states produced by the sector (2.46).

$\bar{\psi}^1$	$\bar{\psi}_c^1$	$\bar{\psi}^2$	$\bar{\psi}_c^2$	$\bar{\psi}^3$	$\bar{\psi}_c^3$	$\bar{\psi}^4$	$\bar{\psi}_c^4$	$\bar{\psi}^5$	$\bar{\psi}_c^5$
–1	–1	–1	–1	0	0	0	0	0	0
–1	–1	0	0	–1	–1	0	0	0	0
–1	–1	0	0	0	0	–1	–1	0	0
–1	–1	0	0	0	0	0	0	–1	–1
0	0	–1	–1	–1	–1	0	0	0	0
0	0	–1	–1	0	0	–1	–1	0	0
0	0	–1	–1	0	0	0	0	–1	–1

Table 2.1 continued.

$\bar{\psi}^1$	$\bar{\psi}^1_c$	$\bar{\psi}^2$	$\bar{\psi}^2_c$	$\bar{\psi}^3$	$\bar{\psi}^3_c$	$\bar{\psi}^4$	$\bar{\psi}^4_c$	$\bar{\psi}^5$	$\bar{\psi}^5_c$
0	0	0	0	–1	–1	–1	–1	0	0
0	0	0	0	–1	–1	0	0	–1	–1
0	0	0	0	0	0	–1	–1	–1	–1
–1	–1	1	1	0	0	0	0	0	0
–1	–1	0	0	1	1	0	0	0	0
–1	–1	0	0	0	0	1	1	0	0
–1	–1	0	0	0	0	0	0	1	1
0	0	–1	–1	1	1	0	0	0	0
0	0	–1	–1	0	0	1	1	0	0
0	0	–1	–1	0	0	0	0	1	1
0	0	0	0	–1	–1	1	1	0	0
0	0	0	0	–1	–1	0	0	1	1
0	0	0	0	0	0	–1	–1	1	1
1	1	–1	–1	0	0	0	0	0	0
1	1	0	0	–1	–1	0	0	0	0
1	1	0	0	0	0	–1	–1	0	0
1	1	0	0	0	0	0	0	–1	–1
0	0	1	1	–1	–1	0	0	0	0
0	0	1	1	0	0	–1	–1	0	0
0	0	1	1	0	0	0	0	–1	–1
0	0	0	0	1	1	–1	–1	0	0
0	0	0	0	1	1	0	0	–1	–1
0	0	0	0	0	0	1	1	–1	–1
1	1	1	1	0	0	0	0	0	0

Table 2.1 continued.

$\bar{\psi}^1$	$\bar{\psi}^1_c$	$\bar{\psi}^2$	$\bar{\psi}^2_c$	$\bar{\psi}^3$	$\bar{\psi}^3_c$	$\bar{\psi}^4$	$\bar{\psi}^4_c$	$\bar{\psi}^5$	$\bar{\psi}^5_c$
1	1	0	0	1	1	0	0	0	0
1	1	0	0	0	0	1	1	0	0
1	1	0	0	0	0	0	0	1	1
0	0	1	1	1	1	0	0	0	0
0	0	1	1	0	0	1	1	0	0
0	0	1	1	0	0	0	0	1	1
0	0	0	0	1	1	1	1	0	0
0	0	0	0	1	1	0	0	1	1
0	0	0	0	0	0	1	1	1	1

It is important to note that the reordering redundancy for the basis vectors does not apply here. Complex right moving charges in a state represent eigenvalues for roots and weights in a representation of a symmetry group. Thus, the order of these eigenvalues will play a role in which groups occur in a model after the GSO projections. Producing charges for a state's left mover is a little more simple. For a state coming from a boson sector defined by (2.39), raising or lowering the space-time fermion modes completely takes care of the mass contribution, as does raising or lowering an internal LM mode. However, raising the internal modes produces ST scalar particles, which are not tabulated in this study. For a state coming from a fermion or SUSY sector, it can be easily shown that raising a periodic or zero mode will make the state massive. Consider the following SUSY sector state:

$$\Big(\big(\tfrac{1}{2}\ \tfrac{1}{2}\big)\ \big(\tfrac{1}{2}\ 0\ 0\big)\ \big(\tfrac{1}{2}\ 0\ 0\big)\ \big(\tfrac{1}{2}\ 0\ 0\big)\ \big(\tfrac{1}{2}\ 0\ 0\big)\ \big(\tfrac{1}{2}\ 0\ 0\big)\ ||\ \vec{0}^{44}\Big)$$

This state already satisfies (2.45), so raising a periodic or zero mode will make it massive. Lowering, however, does the following: $\frac{1}{2} \rightarrow -\frac{1}{2}$, which has no effect on

the state's LM mass squared. Thus, sectors with already massive left movers cannot produce massless states. For the left-right paired real fermion modes, there are a couple of rules to consider.

Apply only the lowering operator to the right moving part of the mode. (2.47)

The RM mass gained by lowering a right moving real mode is weighted the same as lowering a complex pair. (2.48)

Both of the above rules are due to a redundancy in the states produced. Lowering the left moving part of the pair produces a physically equivalent and indistinguishable state as lowering the right moving part of the real fermion pair. (2.47, 2.48) are essentially taking care of the redundancy without explicitly constructing the two states. Consider as an example the following set of eight elements, $(y_1\ y_2\ w_5\ w_6\ ||\ \bar{y}_1\ \bar{y}_2\ \bar{w}_5\ \bar{w}_6)$

$$\left(\frac{1}{2}\ \frac{1}{2}\ \frac{1}{2}\ 0\ ||\ 0\ 0\ \frac{1}{2}\ 0\right) \tag{2.49}$$

The pairings are as follows: $(y_1\ y_2)\ (w_5\ ||\ \bar{w}_5)\ (w_6\ ||\ \bar{w}_6)\ (\bar{y}_1\ \bar{y}_2)$

The possible charge states, then, are:

Table 2.2: Possible charge states produced by (2.49). To make these massless, additional fractional elements from other modes in the states are needed.

y_1	y_2	w_5	w_6	$\bar{y}_1$	$\bar{y}_2$	$\bar{w}_5$	$\bar{w}_6$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	–1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	–1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	1

Table 2.2 continued.

y_1	y_2	w_5	w_6	$\bar{y}_1$	$\bar{y}_2$	$\bar{w}_5$	$\bar{w}_6$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	–1	–1	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	–1	–1	$-\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	1	1	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	1	1	$-\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	–1
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	–1
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	1
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	1
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	–1	–1	$\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	–1	–1	$-\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	1	1	$\frac{1}{2}$	0
$-\frac{1}{2}$	$-\frac{1}{2}$	$\frac{1}{2}$	0	1	1	$-\frac{1}{2}$	0

Tables 2.1, 2.2 shown above are the possible states produced by certain segments of a larger $\vec{\alpha}$. When all possibilities for the 64 component $\vec{\alpha}$ are taken into consideration the number of computations required becomes quite large, as are the number of states produced. This step can take a long time to perform with a brute force computer program algorithm, but luckily some observations allow for a more finessed approach to be possible.

Application of an $\vec{F}$ operator cannot decrease the mass of the state. (2.50)

This is due to the range of the possible values for the elements of $\vec{\alpha}$, given by (2.1). Before $\vec{F}$ is applied, the new range for the values of $\vec{Q}$ is:

$$-\frac{1}{2} < Q_i \leq \frac{1}{2} \tag{2.51}$$

It is easy to see that adding positive or negative one to the element will increase the value of the element squared unless it is equal to $\frac{1}{2}$, in which case lowering will result in the same value of the element squared. This fact can be used to trim the computing time required to produce the states for a model.

2.3.4 GSO Projection

Not all of the states in a model can be used simultaneously. Another degree of freedom granted by modular invariance is that only certain states can be present in a model together. This freedom is embodied by the GSO Projection equation. The GSO Projection (GSOP) is present in all forms of string theory. In WCFFHS models, the states which can simultaneously exist in a model are solutions to the following equation:

$$\frac{1}{2}\vec{\alpha}_j^B \cdot \vec{Q}_{\vec{\alpha}} = \sum_i k_{ji} a_i + s_j \pmod 2, \tag{2.52}$$

where $\vec{Q}_{\vec{\alpha}}$ is a state coming from a sector $\vec{\alpha}$, k_{ji} specifies an element of the GSO coefficient matrix, a_i is the coefficient on the i^{th} $\vec{\alpha}^B$ which produced the sector which produced the state. s_j is, as in equation (2.28), 0 for a space-time boson and 1 for a space-time fermion $\vec{\alpha}_j^B$. The dot product is a Lorentz dot product with a special regard to real fermion modes, coming from the physical state degeneracy mentioned in section 2.3.3.

$$\vec{\alpha}^B \cdot \vec{Q}_{\vec{\alpha}} = \vec{\alpha}_L^B \cdot \vec{Q}_{\vec{\alpha},L} - \vec{\alpha}_R^B \cdot \vec{Q}_{\vec{\alpha},R} + \frac{1}{2}(\vec{\alpha}_{L,real}^B \cdot \vec{Q}_{\vec{\alpha},L,real} - \vec{\alpha}_{R,real}^B \cdot \vec{Q}_{\vec{\alpha},R,real}) \tag{2.53}$$

All states which satisfy equation (2.52) for all j (each $\vec{\alpha}^B$) can exist in a given model. It is clear from equation (2.52) that the degrees of freedom specified by the GSO coefficient matrix can drastically affect an WCFFHS model. There is one

simplification that can be applied to the GSOP for gauge boson states. Since the ST left mover fermion modes are equal to 1, the dot product of the $\vec{\alpha}^B$ with $\vec{Q}$ is equal to the value of s_j, so they cancel. Thus only the right moving dot products need be considered for gauge boson state GSOPs.

$$-\frac{1}{2}\vec{\alpha}^B_{j,R} \cdot \vec{Q}_{\vec{\alpha},R} = \sum_i k_{ji} a_i \pmod 2, \tag{2.54}$$

where the subscript R signifies the right moving part of the vectors. The dot product on the left side of the equation is the same as the dot product defined in equation (2.53), except without left moving parts.

2.4 ST SUSY

WS SUSY is built into WCFFHS models, but ST SUSY is not. The number of ST SUSYs is determined by counting the gravitinos, found by building all of the states from the SUSY generating sector, defined by equation (2.40). For a SUSY sector, only the left moving states need be considered. It may seem as though the right mover isn't massless (the length squared being zero, rather than 4), but for a gravitino the right moving ST boson mode is considered to be raised, thus making it massless. As mentioned earlier, the right moving ST boson modes are used to produce gravitons. This occurs when the ST fermion modes are raised to ± 1. When the left mover comes from a SUSY sector and the right moving ST boson modes are raised, a gravitino is produced. The number of gravitinos in a theory is equal to the number of ST SUSYs.[4] Once the states from the SUSY sector are built, the ones which pass the GSOP become valid gravitino states.

[4] This is true in models of D=10, 6, and 4 large space-time dimensions. For D=8 the number of space-time supersymmetries is not equal to the number of gravitinos, as there are a different number of bosons and fermions in those models. In odd dimensions the rules for constructing matter states (including gravitinos) have not been fully worked out.

The gauge content is determined by states with ST modes equal to ± 1 originating from a bosonic sector (2.39) which pass the GSOPs (2.54). The complex RM modes (not paired with left moving modes) form the charge lattice for the gauge groups. Dot products over the gauge charges are done as follows

$$\vec{Q}_i^{Gauge} \cdot \vec{Q}_j^{Gauge} = \sum_{n=Complex\ RM} Q_{i,n}^{Gauge} \times Q_{j,N}^{Gauge} \tag{2.55}$$

In group theoretic terminology, these states form the adjoint representation of a group, which is in one-to-one correspondence with the generators of the group. The complex modes form the eigenvalues of this representation with the exception of the Cartan generators, which have eigenvalues of zero. These are not needed explicitly in WCFFHS models, so they are not usually built. Additionally, for every positive[5] root, there is an exact negative also in the representation. Thus, all of the group properties of the model can be determined by the positive roots.

Even without the Cartan generators, the adjoint representations are enough to identify the groups making up the model. To do this, the complete list of gauge boson states must be split into sets which are orthogonal, meaning all of the members of each set has a zero dot product (over the gauge charges only) with all of the states of all other sets. Since the gauge groups of most models are tensor products of smaller gauge groups, this allows the smaller groups to be identified.

Lie algebras fall into two categories based on the length squared of the roots (the eigenvectors in the adjoint representation). Groups in which all of the roots have the same length squared are called simply laced gauge groups, and they include $SU(N+1)$, $SO(2N)$, and $E_{6,7,8}$ where N is the rank of the group. Groups in which the roots have two different lengths (squared) are called non-simply laced, and include $SO(2N+1)$, $Sp(2N)$, F_4, and G_2.

[5] Positivity in the root space is determined by convention to mean the sign of the first nonzero element.

These Lie algebras are themselves embedded in Kač-Moody algebras of varying "levels." The full KM algebras have many different properties which distinguish them mathematically from Lie algebras. The level of a KM algebra sets a limit to the dimension of a Lie algebra, with the "usual" dimension of the Lie algebra being at KM level 1. Higher level (above 1) KM algebra embeddings admit representations with dimensions not seen by level-1 Lie algebras in models. Specifically, spin-1/2 adjoints that are not SUSY partners of gauge bosons appear as representations of higher level KM algebra embeddings. Higher level KM algebra embeddings are identified when all of the roots have a shorter length squared than those of other groups in the model. The KM level of a gauge group is determined by

$$L = 2/l^2_{max}, \tag{2.56}$$

where L is the KM level and l^2_{max} is the maximum length squared of the roots in the group's adjoint. Note that a set of roots with maximum length squared of 2 has a Lie algebra embedded in a KM algebra of level 1.

The groups making up the model, once split into mutually orthogonal sets, can then be classified by their KM level and whether or not they are simply laced. Once that is established, the roots can be used to identify the class and rank of the group. The properties of groups useful for identifying the rank and class are listed in Table 2.3. Table 2.3 shows several ways in which the roots can be used to identify the gauge groups using their adjoint representations. An additional fact which aids this is the maximal rank achievable for a particular number of large space-time dimensions. Those are charted in Table 2.4. Because there is a maximal rank achievable, there are a finite number of available non-Abelian gauge groups which can make up a model. This means that in many cases, inverting the formula for the number of nonzero positive roots is a viable means of identification, particularly for A and D class gauge groups. In general, inverting each formula will yield an integral result for the rank of the group only if the correct class's formula is used.

Table 2.3: This table presents information pertinent to the identification of Lie groups from their nonzero roots. From the left, the columns are the Cartan classification, the colloquial name, the dimension of the adjoint representation, the number of nonzero positive roots, the number of positive short roots, and the Dynkin diagram. The Dynkin diagrams give the simple roots and their dot products with other simple roots. Short roots are filled in, while long roots are empty. The lines represent the angles between the simple roots in the root space, which are essentially the normalized dot products. Because of the normalizations, the number of lines directly corresponds to the ratio of lengths-squared of the two roots connected by the lines.

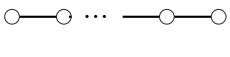
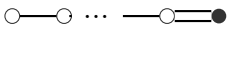
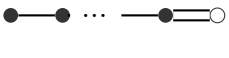
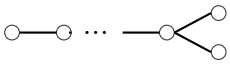
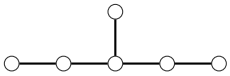
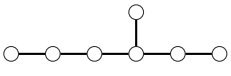
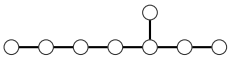
Class	Name	Dim	NZPR	Short	Dynkin Diagram
A_N	$SU(N+1)$	N^2+N+1	$\frac{1}{2}N(N+1)$	0	
B_N	$SO(2N+1)$	$N(2N+1)$	N^2	N	
C_N	$Sp(2N)$	$N(2N+1)$	N^2	$N(N-1)$	
D_N	$SO(2N)$	$N(2N-1)$	$N(N-1)$	0	
E_6	E_6	78	36	0	
E_7	E_7	133	63	0	
E_8	E_8	248	120	0	
G_2	G_2	14	6	3	
F_4	F_4	52	24	12	

Table 2.4: The maximal ranks of the gauge groups based on the number of large space-time dimensions.

Large ST Dimensions	Maximum Gauge Group Rank
10	16
8	18
6	20
4	22

For example, suppose there is a set of 15 nonzero positive roots, all of length squared 2. Because all of the roots are length squared 2, the group is simply laced at KM level 1. Moreover, there are not enough nonzero positive roots to be an E class group, so that leaves only A or D class possible. Inverting the formula for the rank of a D class group in terms of the number of nonzero positive roots, $NZPR$, yields

$$N = \frac{1 + \sqrt{1 + 4 \times NZPR}}{2}. \tag{2.57}$$

Placing 15 into this equation yields a result of ~ 4.41, not an integer. This means the group must be A class. Inverting the formula in Table 2.3 to give the rank in terms of the number of nonzero positive roots $NZPR$, gives the following equation:

$$N = \frac{-1 + \sqrt{1 + 8 \times NZPR}}{2}. \tag{2.58}$$

Setting $NZPR = 15$ gives a result of exactly 5. Thus, the group is A_5, or $SU(6)$.

There are some instances where this method will not work. A few groups with different class and rank which have the same number of nonzero positive roots. They are listed in Table 2.5. Notice in Tables 2.5 and 2.3 that the B_N and C_N groups have exactly the same number of nonzero positive roots. To determine one from the other, counting the number of short roots will suffice. In the other cases shown in Table 2.5, finding the simple roots may be necessary. This can be done by adding and subtracting all of the roots from one another. Any root that can be expressed

Table 2.5: Gauge groups with different ranks and/or classes, but the same number of nonzero positive roots.

Group	Group	NZPR
$SU(9)$	E_6	36
$SU(16)$	E_8	120
$SU(21)$	$SO(30)$	210
$SO(2N+1)$	$Sp(2N)$	N^2

as the sum of other roots is not simple. The dot products between them can also be used to build the Dynkin diagram, which also indicates the rank and class of the group. Notice in Table 2.3 that the simple roots only ever take on negative or zero dot products with another. The largest set of roots with negative or zero dot products between them is the set of simple roots.

While the methods for gauge groups presented in the preceding paragraphs are by no means exhaustive, they are the most common and efficient ways to accomplish this task using a computer program.

Once the non-Abelian gauge groups have been identified, the number of Abelian $U(1)$'s can be determined. This is done by counting the total non-Abelian rank and subtracting it from the total rank. The maximum total ranks for each number of large space-time dimensions are listed in Table 2.4. However, this can be reduced if there are left-right real fermion pairs in the model. The total rank of the gauge group can be found as follows

$$\text{Total Rank} = \text{Max Rank} - \frac{1}{2} \times N_{LR}, \tag{2.59}$$

where N_{LR} is the number of left-right pairs. For clarification, the equation for the number of $U(1)$'s is given below:

$$N_{U(1)} = \text{Max Rank} - \frac{1}{2} \times N_{LR} - \text{Rank}_{NA}. \tag{2.60}$$

2.6 Matter Representations

Once the massless matter states have been determined by the GSO projections, the next task for building the model is to determine the representation dimension of each state under all of the gauge groups. The most efficient way to do this will be identifying the highest weight states.

The weights of a given representation fit into stacks of eigenvalues. Shifting from one to the next involves applying "raising" and "lowering" operators to them. The operators are vectors of eigenvalues of the adjoint representation for the group. The highest weight of a representation is the weight which cannot be raised without leaving the representation. The importance of the highest weight is that there are several ways of mapping it to the dimension of the representation. The two easiest ways will be shown here.

The first is using Dynkin coefficients, sometimes referred to as Dynkin labels. It is a vector of normalized dot products with the simple roots of a group. The vector's elements are built with the equation

$$D_i = \frac{2\vec{\Lambda} \cdot \vec{Q}_i{}^{sr}}{\vec{Q}_i{}^{sr} \cdot \vec{Q}_i{}^{sr}}, \tag{2.61}$$

where D_i is the i^{th} Dynkin coefficient, $\vec{\Lambda}$ is the highest weight, and $Q_i{}^{sr}$ is the charge vector of the i^{th} simple root. Determining whether a state is the highest weight in representation is easy using this method: any weight which has at least one negative Dynkin coefficient is not the highest weight in a representation.

Each set of Dynkin coefficients provides a one-to-one correspondence to a representation. There are some subtleties to this, however. The mapping requires the simple roots to be placed and computed according to the arrangement of the Dynkin diagram. Placing dot products in the incorrect order results in a different matter representation being calculated. Moreover, the Dynkin diagrams themselves have symmetries. This means that there can be relative differences between representations which have the same dimension. Representations that behave in this manner

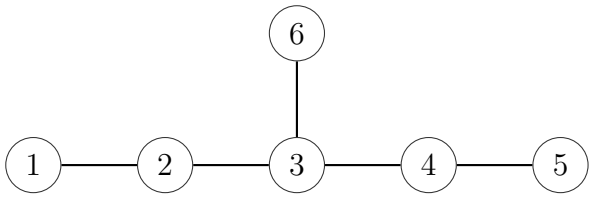


Figure 2.1: The Dynkin diagram for E_6. Notice that exchanging roots 1 and 5 or roots 2 and 4 result in the same relative configurations. Symmetries of this sort allows for complex representations.

are called complex representations. As an example, consider E_6, whose Dynkin diagram is presented in Figure 2.6. Consider the following pair of Dynkin labels.

(1,0,0,0,0,0)

(0,0,0,0,1,0)

The dimension of both representations is 27, yet the overall structure of the two representations is different. Therefore, one is labeled 27, while the other is labeled $\overline{27}$. It doesn't matter which representation is barred since the differences are relative. The bar only serves to distinguish the two when both are present in a model. The caveat to this indistinguishability is that once the convention has been established, an absolute ordering has been placed on the simple roots. All complex representations in a model must follow the same ordering convention. If the Dynkin labels (1,0,0,0,0,0) are chosen to map to the 27 dimensional representation, then the (0,0,0,1,0,0) representation cannot be the 351, it must be the $\overline{351}$, since the Dynkin labels of the 27 fix the ordering. One could also choose (1,0,0,0,0,0) to be the $\overline{27}$, and the (0,0,0,1,0,0) to be the 351.

D_4, or $SO(8)$ has an even greater symmetry for its Dynkin diagram, shown in Figure 2.6. Consider the following three Dynkin labels

(1,0,0,0) (0,0,1,0) (0,0,0,1)

All three of these have dimension 8, yet relative to one another these are distinct representations; one is a vector representation while the others are spinor represen-

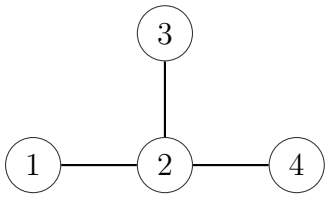


Figure 2.2: The Dynkin diagram for D_4, in which exchanging roots 1, 3, and 4 results in the same relative configurations. This property is called triality.

tations. As with E_6, this only matters if more than one of the three 8's is in the model, up until the ordering of all simple roots has been fixed. Therefore, if the vector rep has been set, but the spinor reps have not, then there is still a degree of freedom in fixing which Dynkin labels map to the spinor, and which map to the conjugate spinor.

The Dynkin diagram method of computing representation dimensions has the advantage of being more intuitive. However, the simple roots making up the diagram must be ordered correctly to obtain the correct result. There is another formula which gives a representation dimension, called the Weyl dimension formula:

$$\dim(\vec{\Lambda}) = \prod_{\vec{Q}_i^r} \frac{(\vec{\Lambda} + \vec{\rho}) \cdot \vec{Q}_i^r}{\vec{\rho} \cdot \vec{Q}_i^r}, \tag{2.62}$$

where $\vec{\Lambda}$ is the weight vector, $\vec{Q}_i^r$ is a nonzero positive root, and $\vec{\rho}$ is the Weyl vector, defined by

$$\vec{\rho} = \frac{1}{2} \sum \vec{Q}_i^r. \tag{2.63}$$

This formula will give the dimension of the representation only if the weight vector $\vec{\Lambda}$ is a highest weight. Otherwise, the product will be either non-integral or zero. While this method does not require knowing the simple roots, it also does not distinguish between the barred and unbarred representations for a group.

Both of these methods are viable ways to determine the dimension of a highest weight for a particular gauge group. For the entire model, the representations must be highest weights of *all* gauge groups to be a highest weight in the model.

2.7 Summary

The model building process begins by specifying the basis vectors for the model, which can be any combination of orders and layers, and the GSO coefficient matrix. Both of these must pass the modular invariance constraints to produce a physically realistic model. If integer codes are used, the basis vectors and GSO coefficient matrix values must be converted into phase values. The fermion phases which are identical for all layers in the model are identified with special attention to left-right pairs. Once that is done, linear combinations of the basis vectors are then built, with coefficients depending on the order. These sectors are then classified based on whether they will produce gravitinos, spin-1 gauge bosons, or spin-1/2 matter fermions. The massless states for each are then built using the raising/lowering operator. Each massless state must pass the GSO projection constraints to remain in the model. The states coming from gravitino generating sectors are counted to determine the number of space-time supersymmetries in the model. Gauge states coming from boson generating sectors are arranged into mutually orthogonal groups, which are identified using the nonzero positive roots. The number of rank-cuts is subtracted from the maximum rank, then the total rank of the non-Abelian gauge groups is subtracted from this to determine the number of $U(1)$ groups. The matter content is identified by finding the dimensions of the highest weight states for all of the gauge groups.

The above process will determine the non-Abelian gauge and matter content, the number of $U(1)$'s, and the number of space-time SUSYs. More analysis must be done for a model to be complete, however. The $U(1)$ charges must be computed next, then the superpotential can be built. With that, D- and F- flat direction vacuum expectation values can be found. These VEVs can then be used to give mass to non-MSSM exotics that were built in the model.

CHAPTER THREE

Challenges in Systematic WCFFHS Searches

Described in this chapter are the challenges to systematic searches of WCFFHS models including model building speed, model data storage, and model comparison. Many of these problems are being addressed through a new framework for WCFFHS model construction. This framework, developed in C++, can be more than 100 times faster than the current FORTRAN 77 software. Moreover, as little as possible has been "hard coded" into the framework- the core logic need not be changed for searches of increasing complexity. This ensures that if the results of a search with few inputs are valid, then the more complex searches will also be valid. It is also object-oriented, allowing it to be easily adapted and expanded for new analyses for future work.

3.1 The Scale of Systematic WCFFHS Searches

The scope of the systematic WCFFHS model searches can be limited by several factors. The first, and most obvious, factor is that of model building speed. Because the number of models to be built can get quite large, the speed at which they can be built is essential. Baylor's FORTRAN 77 software for WCFFHS model building can construct a model in between five and ten seconds. A data set with 1,000,000 models would take about 277 hours, about eleven days. As the number of "layers" of basis vectors in a model increases, the number of total models to build grows exponentially; each of the 1,000,000 models now has on the order of 1,000,000 new basis vectors (likely more than that) with modular invariance. Thus, the total number of models in the data set of the next layer is an estimated 10^{12}. At one model every ten seconds, the layer-2 data set would be completed in approximately 320,000 years.

Several steps can be taken to reduce the time required for model construction, both from an analytical and implementation standpoint. From the implementation side, understanding optimization techniques when writing the model building software is essential. Minimizing file and screen i/o, as well as the amount of copying done in memory are crucial steps needed to reduce the computing time. This idea has been central to the construction of the C++ framework. Use of a profiler for finding key "choke points" in the model building process has aided in the implementation of these principles. The C++ framework can build between 10 and 20 models per second with the compiler optimization flags activated, more than 100 times faster than the FORTRAN 77 software. Not yet implemented are techniques for distributed computing. Use of multinode processing can also speed up systematic searches. These features have yet to be added to the framework, but will be incorporated at a later time.

Another computational concern is one of data storage. While a "master atlas" of basis vectors and the models to which they map is desirable, it is not feasible. The estimated average amount of space per model is 0.725 kB. For the 1,000,000 model data set, this amounts to 725 MB. The next layer would use approximately 725×10^3 TB, making a pure "atlas" of WCFFHS models unreasonable to pursue. There are two solutions to this problem. One is to gather statistics as the models are generated without keeping the models themselves in memory. While this approach is less taxing on system resources, it does have disadvantages. In particular, one must know the statistics to be gathered on the models prior to runtime. Any additional statistics would require a second run. Moreover, the total model set would have to be statistically analyzed, as opposed to the distinct models only, as the models are not being kept in memory. Double-counts that result from the construction method itself cannot be avoided with this approach. The other approach, implemented in the C++ framework, is to count the distinct models only, writing those to a file

that serves as a small repository. While a full discussion of uniqueness in WCFFHS models is discussed in the next section, it suffices to say there are disadvantages to this approach as well. In particular, the definition of uniqueness in these models is somewhat nebulous. Additionally, each unique model that is found must be compared with the other unique models that have been constructed, making each model require more computing time to complete.

Analytic techniques can also be used to reduce the number of redundant models produced. As these techniques are related to uniqueness in WCFFHS phenomenology, discussion of such analysis will be deferred until after uniqueness has been addressed.

3.2 Uniqueness in WCFFHS Models

The previously described computational limitations force the analysis to be only on models that are considered distinct. The definition of distinctness amongst WCFFHS models is not clear. In particular the "amount" of phenomenology done to distinguish the models from one another must be balanced with the amount of computing time required to perform such analyses, as well as how easily the analyses can be automated.

Consider two different basis vector sets producing the same gauge and non-Abelian matter content. Even though the actual charge vectors and gauge group eigenvalues for the two models are different, the overarching group structures are identical. This could be due to a string-scale type symmetry within the construction method itself, in which case there would be some sort of transformation that could be applied to one of the basis vector sets to reproduce the other. The $U(1)$ charges of those two models, once diagonalized, would be the same. Such a symmetry in the construction method could in principle be revealed through an analytic study. However, it could also be the case that these models have completely different $U(1)$

Table 3.1: Two models illustrating the inherent difficulty in comparing WCFFHS models.

QTY	$SU(4)$	$SU(4)$	$SU(4)$	$SO(10)$	E_8
1	4	4	1	1	1
1	4	1	4	1	1
1	1	4	4	1	1
2	1	6	1	10	1
2	1	1	6	10	1

QTY	$SU(4)$	$SU(4)$	$SU(4)$	$SO(10)$	E_8
1	4	4	1	1	1
1	4	1	4	1	1
1	1	4	4	1	1
2	6	1	1	10	1
2	1	1	6	10	1

structures, and that the GSOPs projected out the states that were different between the models. Such models will not have identical superpotentials or D- and F-flat directions, all of which have significant impact on the phenomenological viability of a model. For the systematic searches in this study the $U(1)$ charges of the matter representations will not be considered. Thus, two models will be considered identical if they have the same gauge and non-Abelian matter content.

This approach has a caveat, however. As the models become increasingly complex, comparing two models becomes increasingly difficult to automate. Consider the following two "toy" models whose particle content is presented in Tables 3.1. It is clear that these two models are equivalent if two of the $SU(4)$ groups are switched. However, a simple boolean comparison of the gauge groups and matter states by a computer program will result in these models being counted as distinct. The root of the problem lies in the fact that the three $SU(4)$ gauge groups are not identical — there are different matter representations that transform under them. The most obvious solution would be to perform brute force permutations on the identical gauge groups and resorting the matter representations. Such an approach would work,

Table 3.2: Matter representation classes of the "toy" models.

QTY	Model 1 Classes	Model 2 Classes
	$SU(4), SU(4)$	
1	(4,4)	(4,4)
1	(4,4)	(4,4)
1	(4,4)	(4,4)
	$SU(4), SO(10)$	
2	(6,10)	(6,10)
2	(6,10)	(6,10)

but takes a significant amount of computing time. Comparisons are the most called operation in a systematic search, however, and thus need to be as fast as possible in order for the search to be efficient.

The solution implemented in this study is to propose and use a conjecture that defines uniqueness in a slightly stronger way. The conjecture is that identical models will have matter that fits into the same "classes" of representations. These matter representation classes are formed by removing the singlets and looking only at the dimension of the representations and the groups under which they transform. The classes of matter representations for the two example models are presented in Table 3.2. Now it is clear the two models are equivalent.

To prove the conjecture, one would need to show that modular invariance prevents models that have only one representation switched with another. Table 3.3 shows an example of such a model.

The model presented in Table 3.3 would be counted as identical to the models in Table 3.1, as it has the same classes of matter representations. However, it could not satisfy modular invariance according to the conjecture. More theoretical work will need to be done to prove or disprove this conjecture.

Table 3.3: Another "toy" model, declared non-existant by the conjecture about matter representation classes.

QTY	$SU(4)$	$SU(4)$	$SU(4)$	$SO(10)$	E_8
2	4	4	1	1	1
1	4	1	4	1	1
2	1	6	1	10	1
2	1	1	6	10	1

The conjecture does not fully remedy the problem of a preferred gauge group ordering. Generally, matter representations are limited by the charges they carry due to the masslessness constraints of the fermion states. Representations of larger dimension tend to carry smaller dimensional charges under the other groups in the model, if at all. Certain gauge groups produce representations of similar dimension, however, which still place an ordering on the gauge groups within a class of matter representation. In particular, the 4- and 5-dimensional representations of $SO(5)$, the 4- and 6- dimensional representations of $SU(4)$, and the 2- and 3-dimensional representations of $SU(2)^{(2)}$ can still cause double counting amongst models. To completely remove the dependence on ordering in WCFFHS models, the problem must be reduced to a counting problem. In addition to proving or disproving the conjecture, the following possibility could also be explored.

Two models with identical gauge groups and ST SUSYs are identical if they have the same total number of matter representation classes, the same number of distinct matter representation classes, and the same number of total fermions.

This would remove the ordering dependence from the model comparison. Until it is proven, however, there is a risk of undercounting the models. Rather than risking this undercounting with the systematic studies presented herein, models in smaller data sets were examined, and duplicates were removed by hand. This will provide a systematic uncertainty estimate for statistics from larger data sets.

CHAPTER FOUR

Redundancies in Explicitly Constructed Ten Dimensional Heterotic String Models

The first of several systematic surveys of the heterotic string landscape presented is one in which models with ten large ST dimensions were constructed. As they have fewer fermionic degrees of freedom, the searches are simpler and take less computing time. Moreover, these models are well known and have been constructed before. This serves as an error check for the WCFFHS model building software used for this study, the FF Framework. The goal of the examination to follow is to search for patterns that may lead to redundancies in the construction method itself. By studying how the basis vector inputs map to the particle content outputs, conclusions can be drawn regarding the WCFFHS method's redundancies. Ultimately, this information will be used for more efficient systematic searches.

4.1 $D = 10$ Heterotic String Models in the Free Fermionic Construction

The spectrum of $D = 10$ heterotic string models has been well tabulated [55]. All of these are presented in detail in Table 4.1, except for an additional heterotic model containing a single E_8 at Kač-Moody level 2. In the free fermionic construction, the latter model is constructed using real Ising fermions [1] [56, 57, 58, 59, 54, 60, 61, 62, 63, 64] , which have not been included in the present study. The emphasis of the discussion to follow will be on the different manifestations of the construction parameters that produced the $D = 10$, level-1 models. Such an examination will provide clues as to the redundancies inherent to the free fermionic heterotic construction, and will aid in more efficient systematic examinations of the input parameters for models with fewer large space-time dimensions.

[1] Higher level Kač-Moody algebras are possible with compact dimensions using left-right pairings, sometimes referred to as rank-cutting. For models in this study, all modes are paired as complex modes on the left and right moving parts of the basis vectors. Thus, higher level Kač-Moody algebras will not appear in these data sets.

Table 4.1: These are all possible $D = 10$, level-1 models that can be constructed using the methods detailed. The dimensions of the non-Abelian matter representations are given underneath the respective gauge groups under which they transform. Abelian charges were not computed.

Model	ST SUSY	Model	ST SUSY
$SO(32)$	1	$SO(32)$	0
$E_8 \otimes E_8$	1	$SO(16) \otimes SO(16)$ $1 \quad 128$ $128 \quad 1$ $16 \quad 16$	0
$SO(8) \otimes SO(24)$ $\overline{8} \quad 24$ $8 \quad 24$	0	$SO(16) \otimes E_8$ $\overline{128} \quad 1$ $128 \quad 1$	0
$SU(2) \otimes SU(2) \otimes E_7 \otimes E_7$ $1 \quad 2 \quad 1 \quad 56$ $1 \quad 2 \quad 56 \quad 1$ $2 \quad 1 \quad 1 \quad 56$ $2 \quad 1 \quad 56 \quad 1$	0	$SU(16) \otimes U(1)$ $\overline{120}$ $\overline{120}$ 120 120	0

All of the searches have the following parameters fixed:

- The first basis vector, $\vec{\mathbb{1}}$, in each model is completely made up of periodic modes ($\vec{1}^{\,8}||\vec{1}^{\,32}$), and serves as a canonical basis for the vectors. It is not shown when describing the input for a model.

- The four left moving complex world-sheet fermions $\psi_c^1, ..., \psi_c^4$ (in light cone gauge) with space-time indices have all periodic boundary conditions and may contribute to the matter states in the model as well as the gauge states. The orders specified will be for the right moving part of the basis vector only, the left moving part will always be order 2. Thus, the total order remains N for N_{even}, and $2N$ for N_{odd}.

- The number of ST SUSYs is found by counting gravitinos [2] .

4.2 Searches With One Basis Vector

For the searches with a single basis vector (beyond the all-periodic), the order was increased with all possible basis vectors of that order investigated. Figure 4.1 shows the number of unique models vs the order of the basis vectors in the search. Notice that for odd orders, there are only two unique models constructed. For each of the odd orders, the models are $SO(32)$ and $E_8 \otimes E_8$, both with $N = 1$ space-time supersymmetry. These are the only supersymmetric models present in the D=10 heterotic landscape.

ST SUSY in WCFFHS models, as mentioned in section 4.1, is determined by counting gravitinos. Computationally, this involves first picking out all gravitino generating sectors as linear combinations of the basis vectors. This sector is identified as having a massless left moving part and all zeros for the right moving part. The only sector with this property is ($\vec{1}^{\,8}||\vec{0}^{\,32}$) (in ten large space-time dimensions). This sector is produced from every basis vector with odd order due to the coefficients that

[2] This method does not give exact results for D=8 models.

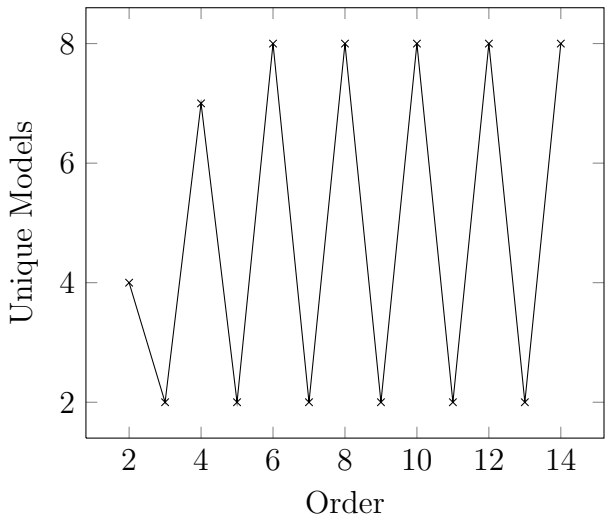


Figure 4.1: Plotted here are the number of unique models produced vs the order of the basis vectors that produced those models.

multiply it. For example, an order 3 basis vector has allowable phases of $0,\ \frac{2}{3},\ -\frac{2}{3}$ on the right side, and phases of $0, 1$ on the left. Thus, the total order of the basis vector is $LCM(2,3) = 6$. The coefficients that multiply that basis vector when constructing the sectors are 0 through 5. The right moving side has a $\mathbb{Z}_3$ symmetry, which means that the values go through the following transformations:

$$\pm\left(\frac{2}{3}\right) \xrightarrow{\times 0} 0 \xrightarrow{\times 1} \pm\left(\frac{2}{3}\right) \xrightarrow{\times 2} \mp\left(\frac{2}{3}\right) \xrightarrow{\times 3} 0 \xrightarrow{\times 4} \pm\left(\frac{2}{3}\right) \xrightarrow{\times 5} \mp\left(\frac{2}{3}\right) \tag{4.1}$$

The left moving side of the basis vector, however, has a $\mathbb{Z}_2$ symmetry, and its values go though the following transformations:

$$1 \xrightarrow{\times 0} 0 \xrightarrow{\times 1} 1 \xrightarrow{\times 2} 0 \xrightarrow{\times 3} 1 \xrightarrow{\times 4} 0 \xrightarrow{\times 5} 1 \tag{4.2}$$

Because the basis vector elements have the $\mathbb{Z}_2^L||\mathbb{Z}_3^R$ symmetry rather than a purely $\mathbb{Z}_6$ symmetry, the gravitino generating sector always emerges. More generally, for any odd right moving order N, the basis vector (and consequently the associated sectors) has a $\mathbb{Z}_2^L||\mathbb{Z}_N^R$ symmetry, rather than a purely $\mathbb{Z}_{LCM(2,N)}$ symmetry. Therefore all basis vectors with odd right moving orders will produce a gravitino generating sector.

We now conjecture that any model of even dimension with a single basis vector of odd order and massless left mover has the maximal number of space-time SUSYs. Models matching these conditions are presented in Table 4.2. The table clearly shows this conjecture to be true for $D = 4$ models and $D = 6$ models meeting these criteria. The conjecture was tested for order-5 models in $D = 4$ and $D = 6$ large space-time dimensions as well, with a sample of the relevant models (containing a massless left mover only) presented in Table 4.3. All of those models (and the ones not shown explicitly) contain the maximal number of space-time supersymmetries.

To prove this conjecture, one must consider how the gravitinos are created from the sector. Applying the raising and lowering operator, $\vec{F}$, according to the following equation generates possible gravitinos:

$$\vec{Q} = \frac{1}{2}\vec{\alpha} + \vec{F}, \tag{4.3}$$

where $\vec{Q}$ is the state, $\vec{\alpha}$ is the sector that generated the state, and $\vec{F}$ is the raising/lowering operator. $\vec{F}$ is any vector consisting of $0, \pm 1$ such that the state is massless[3] . The GSO projections will then choose which possible states generated by the $\vec{F}$'s in equation (4.3) is physically present in the model. The equation for the GSO projections is

$$\vec{v}_j \cdot \vec{Q}_{\vec{\alpha}} = \sum_i a_i k_{ji} + s_j \pmod 2, \tag{4.4}$$

where $\vec{Q}_{\vec{\alpha}}$ is the state coming from the sector $\vec{\alpha}$, a_i are the coefficients that produced the sector $\vec{\alpha}$, k_{ji} are elements of the GSO coefficient matrix, and s_i is equal to one if $\vec{v}_j$ is a space-time fermion sector, and 0 if $\vec{v}_j$ is a space-time boson sector. The dot product in equation (4.4) and the other equations in this section is a Lorentz dot product, where the dot products of the right movers are subtracted from the dot

[3] With the modes expressed in a complex basis, The conditions for masslessness are that the length squared of the left mover be equal to 1, while the length squared of the right mover be equal to 2.

Table 4.2: Order-3 models with six and four large space-time dimensions and massless left movers. This table provides evidence for a conjecture that single basis vectors with odd order right movers always have the maximal number of space-time supersymmetries. Note also that only half of the k_{ij} matrix is specified. The other half is constrained by modular invariance, and is therefore not a true degree of freedom for WCFFHS models. The basis vectors are presented in a real basis.

D	BV	k_{ij}	Model	N
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,34}(\frac{\vec{2}}{3})^6)$	$(\frac{1}{0})$	$SO(40)$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,22}(\frac{\vec{2}}{3})^{18})$	$(\frac{1}{0})$	$SO(24) \otimes E_8$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,16}(\frac{\vec{2}}{3})^{24})$	$(\frac{1}{0})$	$SO(16) \otimes SO(24)$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,10}(\frac{\vec{2}}{3})^{30})$	$(\frac{1}{0})$	$SU(16) \otimes SO(10)$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,4}(\frac{\vec{2}}{3})^{36})$	$(\frac{1}{0})$	$SU(2)^3 \otimes SU(18)$	2
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,38}(\frac{\vec{2}}{3})^6)$	$(\frac{1}{0})$	$SO(44)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,26}(\frac{\vec{2}}{3})^{18})$	$(\frac{1}{0})$	$SO(28) \otimes E_8$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,20}(\frac{\vec{2}}{3})^{24})$	$(\frac{1}{0})$	$SO(20) \otimes SO(24)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,12}(\frac{\vec{2}}{3})^{30})$	$(\frac{1}{0})$	$SU(16) \otimes SO(14)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,8}(\frac{\vec{2}}{3})^{36})$	$(\frac{1}{0})$	$SU(2) \otimes SU(18) \otimes SO(8)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,2}(\frac{\vec{2}}{3})^{42})$	$(\frac{1}{0})$	$SU(21) \otimes U(1) \otimes U(1)$	4

Table 4.3: A sample of order-5 models with six and four large space-time dimensions and massless left movers. All of them have the maximal number of space-time supersymmetries. The basis vectors are presented in a real basis.

D	BV	k_{ij}	Model	N
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,12}(\vec{\frac{2}{5}})^{14}(\vec{\frac{4}{5}})^{14})$	$(\frac{1}{0})$	$SO(12)\otimes E_7\otimes E_7$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,8}(\vec{\frac{2}{5}})^{16}(\vec{\frac{4}{5}})^{16})$	$(\frac{1}{0})$	$SO(8)\otimes SO(16)\otimes SO(16)$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,6}(\vec{\frac{2}{5}})^{32}(\vec{\frac{4}{5}})^{2})$	$(\frac{1}{0})$	$SO(8)\otimes SO(32)$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,6}(\vec{\frac{2}{5}})^{32}(\vec{\frac{4}{5}})^{12})$	$(\frac{1}{0})$	$SU(4)\otimes SU(12)\otimes E_6$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,4}(\vec{\frac{2}{5}})^{18}(\vec{\frac{4}{5}})^{18})$	$(\frac{1}{0})$	$SU(2)^2\otimes SO(10)^2$	2
6	$(\vec{1}^{\,4}(1,0,0)^4 \| \vec{0}^{\,2}(\vec{\frac{2}{5}})^{24}(\vec{\frac{4}{5}})^{14})$	$(\frac{1}{0})$	$SU(12)\otimes SO(14)\otimes U(1)^2$	2
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,16}(\vec{\frac{2}{5}})^{14}(\vec{\frac{4}{5}})^{14})$	$(\frac{1}{0})$	$SO(16)\otimes E_7\otimes E_7$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,10}(\vec{\frac{2}{5}})^{22}(\vec{\frac{4}{5}})^{12})$	$(\frac{1}{0})$	$SU(12)\otimes SO(10)\otimes E_6$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,8}(\vec{\frac{2}{5}})^{18}(\vec{\frac{4}{5}})^{18})$	$(\frac{1}{0})$	$SU(10)\otimes SU(10)\otimes SO(8)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,6}(\vec{\frac{2}{5}})^{24}(\vec{\frac{4}{5}})^{14})$	$(\frac{1}{0})$	$SU(4)\otimes SU(12)\otimes SO(14)\otimes U(1)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,4}(\vec{\frac{2}{5}})^{30}(\vec{\frac{4}{5}})^{10})$	$(\frac{1}{0})$	$SU(2)^2\otimes SU(16)\otimes SO(10)$	4
4	$(\vec{1}^{\,2}(1,0,0)^6 \| \vec{0}^{\,4}(\vec{\frac{2}{5}})^{20}(\vec{\frac{4}{5}})^{20})$	$(\frac{1}{0})$	$SU(2)^3\otimes SU(10)\otimes SU(10)$	4

products of the left movers. Moreover, complex modes contribute twice as much to the dot product as real modes.

In the case of gravitinos, the mass shell condition for the right movers is ignored. For gravitinos, the space-time boson modes customarily left out of this construction method to save computing resources are raised. When the space-time boson modes are raised, none of the other right moving modes can be raised without giving the state mass.

The masslessness condition for the left movers is already fulfilled when the SUSY generating sector is produced. Since the raising operators will make the left mover massive and the lowering operators do not change the mass, only the lowering operators are applied. Lowering the space-time fermion modes will change the spin state of the same gravitino (within its given helicity). Only one gravitino helicity is allowed by the GSO projections per model in ten dimensions, while both can be present in dimensions lower than ten. Lowering the compactified modes (in models with less than ten large space-time dimensions) will create distinguishable gravitino states. This is the reason that D=6 models have $N = 2$ space-time SUSY, while D=4 models have $N = 4$.

The possible gravitino states can be categorized by which compact fermion modes (and space-time modes for D=10) have been lowered with the $\vec{F}$ operator. These fall into one of two categories based on their dot products in the GSO projections (4.4). The possible gravitino states for $D = 10$, 6, and 4 are listed in Table 4.4. The GSO projection equations will reveal why models with a single odd ordered basis vector and a massless left mover always have the maximal number of space-time SUSYs. For possible gravitino states, the equation (4.4) can be simplified.

- The coefficients producing the sector for this class of model is always $(0, N_R)$ where N_R is the order of the right mover.

Table 4.4: The possible gravitino states in 10, 6, and 4 large space-time dimensions. A + represents a charge value of $\frac{1}{2}$, while a – represents a charge value of $-\frac{1}{2}$. The dot products for both of the GSOP constraints are in this case identical. Note that the $y^i,\ w^i$ values can also be periodic and thus can vary, but permutations of $x^i,\ y^i,\ w^i$ produce identical models when there is only one basis vector with a massless left mover. The states are presented in a complex basis.

D	ψ^μ	x^i	Dot
10	+ + + +		0
10	+ + + –		1
6	+ +	+ +	0
6	+ +	+ –	1
6	+ +	– +	1
6	+ +	– –	0
4	+	+ + +	0
4	+	+ + –	1
4	+	+ – +	1
4	+	– + +	1
4	+	+ – –	0
4	+	– + –	0
4	+	– – +	0
4	+	– – –	1

- s_j is equal to 1 since the only basis vectors producing the model are $\vec{\mathbb{1}}$ and $\vec{v}$.

The conjecture can be shown heuristically by noting that with only two possible dot product values and two possible values for $k_{\vec{\mathbb{1}}\vec{v}}$, the space of potential gravitino states can be divided into two parts. This gives a maximum space-time SUSY of $N = 1$ for $D = 10$, $N = 2$ for $D = 6$, and $N = 4$ for $D = 4$, which is known to be true. It can, with some effort, also be proven mathematically.

The GSOPs with the above conditions applied are

$$0 = N_R k_{\vec{\mathbb{1}}\vec{v}} + 1 \pmod 2, \tag{4.5}$$

$$0 = N_R k_{\vec{v}\vec{v}} + 1 \pmod 2, \tag{4.6}$$

for "even" gravitinos (dot product is equal to zero mod 2). For "odd" gravitinos (dot product equal to one mod 2) the GSOPs are

$$1 = N_R k_{\vec{\mathbb{1}}\vec{v}} + 1 \pmod 2, \tag{4.7}$$

$$1 = N_R k_{\vec{v}\vec{v}} + 1 \pmod 2. \tag{4.8}$$

This constrains the possible values that $k_{\vec{\mathbb{1}}\vec{v}}$ and $k_{\vec{v}\vec{v}}$ can take in order for the possible gravitinos to survive the GSO projections.

$$N_R k^e_{\vec{\mathbb{1}}\vec{v}} = N_R k^e_{\vec{v}\vec{v}} = 1 \pmod 2, \tag{4.9}$$

$$N_R k^o_{\vec{\mathbb{1}}\vec{v}} = N_R k^o_{\vec{v}\vec{v}} = 0 \pmod 2, \tag{4.10}$$

where k^e_{ij} is the GSO coefficient required for the even gravitino to pass, and k^o_{ij} is the GSO coefficient required for odd gravitinos to pass. These are the conditions that must be proven to show that basis vectors with a massless left mover and odd ordered right mover always produce the maximal number of space-time supersymmetries. The conditions (4.9, 4.10) imply that choosing $k_{\vec{v}\vec{\mathbb{1}}}$ only ever eliminates half the total possible states, giving the model the maximal number of gravitinos.

The proof will proceed as follows. The conditions (4.9, 4.10) will be rewritten in terms of the order-2 GSO coefficient $k_{\vec{v}\vec{\mathbb{1}}}$ using the modular invariance constraints for the GSO coefficient matrix

$$N_j k_{ij} = 0 \pmod 2, \tag{4.11}$$

$$k_{ij} + k_{ji} = \frac{1}{2}\vec{v}_i \cdot \vec{v}_j \pmod 2, \tag{4.12}$$

$$k_{ii} + k_{i1} = \frac{1}{4}\vec{v}_i \cdot \vec{v}_i - s_i \pmod 2, \tag{4.13}$$

where s_i is defined as in equation (4.4). This places a condition on the dot products of the basis vectors $\vec{\mathbb{1}}$ and $\vec{v}$. A contradiction will be assumed and proven to be logically inconsistent, thus proving the following generalization of (4.9, 4.10)

$$N_R k_{\vec{\mathbb{1}}\vec{v}} = N_R k_{\vec{v}\vec{v}} \pmod 2 \tag{4.14}$$

which is sufficient to prove the conjecture.

Applying the modular invariance conditions to the above conditions results in

$$N_R k_{\vec{\mathbb{1}}\vec{v}} = -N_R k_{\vec{v}\vec{\mathbb{1}}} - \frac{N_R}{2}\vec{\mathbb{1}} \cdot \vec{v} \pmod 2, \tag{4.15}$$

$$N_R k_{\vec{v}\vec{v}} = -N_R k_{\vec{v}\vec{\mathbb{1}}} - \frac{N_R}{4}\vec{v} \cdot \vec{v} - N_R \pmod 2. \tag{4.16}$$

Combining these conditions implies

$$-\frac{N_R}{2}\vec{\mathbb{1}} \cdot \vec{v} = -\frac{N_R}{4}\vec{v} \cdot \vec{v} - N_R \pmod 2. \tag{4.17}$$

Note that N_R (*mod* 2) is one. Modular invariance of basis vectors constrains the possible values for the dot products in the above equation. These constraints in general are

$$N_{ij}\vec{v}_i \cdot \vec{v}_j = 0 \pmod 4, \tag{4.18}$$

$$N_{ii}\vec{v}_i \cdot \vec{v}_i = 0 \pmod 8 \text{ (for even N)}, \tag{4.19}$$

where N_{ij} is the least common multiple of the orders of the basis vectors $\vec{v}_{i,j}$. For the cases being considered, this implies

$$\begin{aligned} 2N_R\vec{\mathbb{1}}\cdot\vec{v} &= 0 \pmod 4 \implies & (4.20)\\ N_R\vec{\mathbb{1}}\cdot\vec{v} &= 0 \pmod 2, & (4.21)\\ 2N_R\vec{v}\cdot\vec{v} &= 0 \pmod 8 \implies & (4.22)\\ N_R\vec{v}\cdot\vec{v} &= 0 \pmod 4. & (4.23) \end{aligned}$$

Thus, the dot product terms in (4.17) are integral. A contradiction can be used since either side can only be zero or one.

Now the faulty assumption will be applied. Assume, instead of (4.17), that the following is true

$$\frac{N_R}{2}\vec{\mathbb{1}}\cdot\vec{v} = \frac{N_R}{4}\vec{v}\cdot\vec{v} \pmod 2. \tag{4.24}$$

Splitting the Lorentz dot products into left and right movers, this condition can be further reduced by noting that $\vec{\mathbb{1}}_L\cdot\vec{v}_L = \vec{v}_L\cdot\vec{v}_L = 4$. In terms of the right movers only, the equation becomes

$$N_R - \frac{N_R}{2}\vec{\mathbb{1}}_R\cdot\vec{v}_R = -\frac{N_R}{4}\vec{v}_R\cdot\vec{v}_R \pmod 2, \tag{4.25}$$

where $N_R = 1 \pmod 2$. The basis vector $\vec{v}$ can be split into its numerator and denominator.

$$\begin{aligned} 1 - \frac{N_R}{2N_R}\vec{\mathbb{1}}_R^N\cdot\vec{v}_R^N &= -\frac{N_R}{4N_R^2}\vec{v}_R^N\cdot\vec{v}_R^N \pmod 2 \implies\\ 1 - \frac{1}{2}\vec{\mathbb{1}}_R^N\cdot\vec{v}_R^N &= -\frac{1}{4N_R}\vec{v}_R^N\cdot\vec{v}_R^N \pmod 2. & (4.26) \end{aligned}$$

The denominator of the right mover is N_R due to the order of the right movers. Additionally, the numerators are all even integers. This comes from the constraint

$$N_R\vec{v}_R = 0 \pmod 2. \tag{4.27}$$

The dot products can now be written as a summation.

$$\begin{aligned} 1 - \frac{2}{2}\sum_i m_i &= -\frac{4}{4N_R}\sum_i m_i^2 \pmod 2 \implies \\ 1 - \sum_i m_i &= \frac{1}{N_R}\sum_i m_i^2 \pmod 2, \end{aligned} \tag{4.28}$$

where m_i is an integer and the sum is over the indices of the right mover. Changing the basis to a multiplicative one, in which the value of the basis vector element is the index and the number of elements with that value, N_i is summed over, we have

$$\begin{aligned} 1 - \sum_{i=1}^{\lfloor N_R/2 \rfloor} iN_i &= \frac{1}{N_R}\sum_{i=1}^{\lfloor N_R/2 \rfloor} i^2 N_i \pmod 2 \implies \\ 1 &= \frac{1}{N_R}\sum_{i=1}^{\lfloor N_R/2 \rfloor} i(i-N_R)N_i \pmod 2. \end{aligned} \tag{4.29}$$

Considering a case by case basis, assume i is even. This means $i(i - N_R)N_i$ is even. If i is odd, then $(i - N_R)$ is even, so $i(i - N_R)N_i$ is also even. Therefore every term in the sum is even, so the total sum is even. This implies that if the sum is a multiple of N_R, it is an even multiple of N_R, and the right hand side of (4.29) is 0 (mod 2) which is logically inconsistent since the left hand side is 1 (mod 2). If the sum is not a multiple of N_R, then it is a non-integral rational number that is also not equal to 1 (mod 2). Therefore (4.29) is impossible to satisfy, which shows the faulty assumption (4.24) is logically impossible. This means (4.17) is true, and the conditions for each (odd or even) gravitino sectors are always satisfied. This proves the conjecture.

Moreover, it can be shown that one choice of $k_{\vec{v}\vec{\mathbb{1}}}$ selects the even gravitinos, while the other choice selects the odd. Consider the modular invariance condition (4.12) applied to this scenario

$$k_{\vec{\mathbb{1}}\vec{v}} + k_{\vec{v}\vec{\mathbb{1}}} = \frac{1}{2}\vec{\mathbb{1}} \cdot \vec{v} \pmod 2. \tag{4.30}$$

By (4.11), $k_{\vec{v}\vec{\mathbb{1}}}$ can only have a value of 0 or 1, and $k_{\vec{\mathbb{1}}\vec{v}}$ is a rational number with denominator N_R. Therefore evenness of the numerator determines whether $N_R k_{\vec{\mathbb{1}}\vec{v}}$

(mod 2) is 0 or 1, passing either the odd or even gravitinos. Using this information, it is clear that the two choices for $k_{\vec{v}\vec{\mathbb{1}}}$ yield

$$\frac{k^N_{\vec{\mathbb{1}}\vec{v}}}{N_R} = \frac{1}{2}\vec{\mathbb{1}}\cdot\vec{v} \pmod 2, \tag{4.31}$$

$$\frac{k^N_{\vec{\mathbb{1}}\vec{v}} + N_R}{N_R} = \frac{1}{2}\vec{\mathbb{1}}\cdot\vec{v} \pmod 2, \tag{4.32}$$

where k^N_{ij} is the numerator of the GSO coefficient. Noting that the right hand side of each equation is an integer, combining them leads to

$$\frac{1}{N_R}(k^{N0}_{\vec{\mathbb{1}}\vec{v}} + k^{N1}_{\vec{\mathbb{1}}\vec{v}} + 1) = 0 \pmod 2, \tag{4.33}$$

with the superscript indicating the choice of $k_{\vec{v}\vec{\mathbb{1}}}$. This equation is only satisfied if the quantity in parentheses is an even multiple of N_R. This is true if and only if $k^{N0}_{\vec{\mathbb{1}}\vec{v}} \neq k^{N1}_{\vec{\mathbb{1}}\vec{v}} \pmod 2$, which means

$$N_R k^{N0}_{\vec{\mathbb{1}}\vec{v}} \neq N_R k^{N1}_{\vec{\mathbb{1}}\vec{v}} \pmod 2. \tag{4.34}$$

Ergo, choosing $k_{\vec{v}\vec{\mathbb{1}}}$ admits either the even or the odd gravitinos into the model.

In addition to the conjecture above, there are also models in each of the four data sets with the exact same particle content without supersymmetry. The basis vectors generating these models have the same right movers, but massive left movers[4] . This removes the gravitino generating sector entirely from the model, leaving the model without supersymmetry. This is shown in Figure 4.2, where the number of unique models is plotted against the number of space-time supersymmetries. The implications of the conclusions made in this section indicate that there is no correlation between the number of space-time supersymmetries and the gauge content for this class of models. More complicated models will need to be tested to determine whether such a correlation will emerge.

[4] This does not happen for D=10, since there are not enough left moving modes to give the potential gravitino generating sector mass.

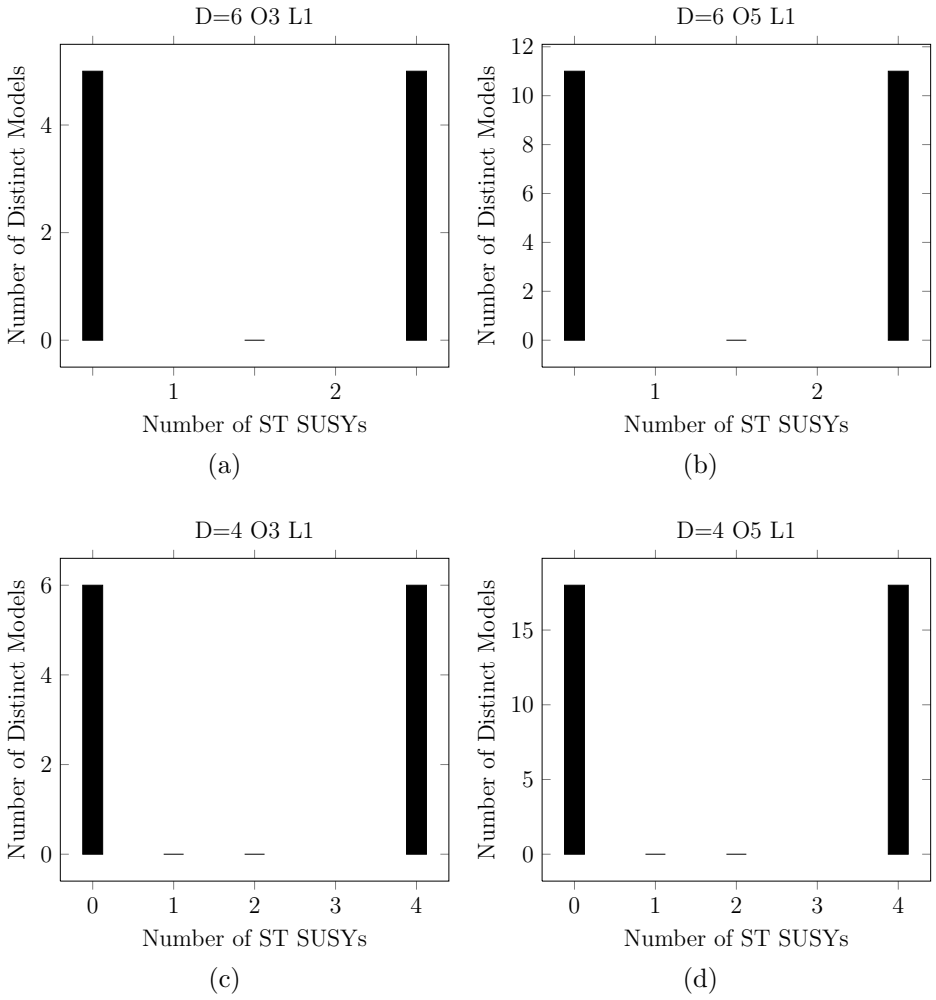


Figure 4.2: The number of distinct models against the number of space-time supersymmetries for $D = 6{,}4$ O3,O5 L1. The number of distinct models with and without space-time SUSY are equal. The models themselves are also equal.

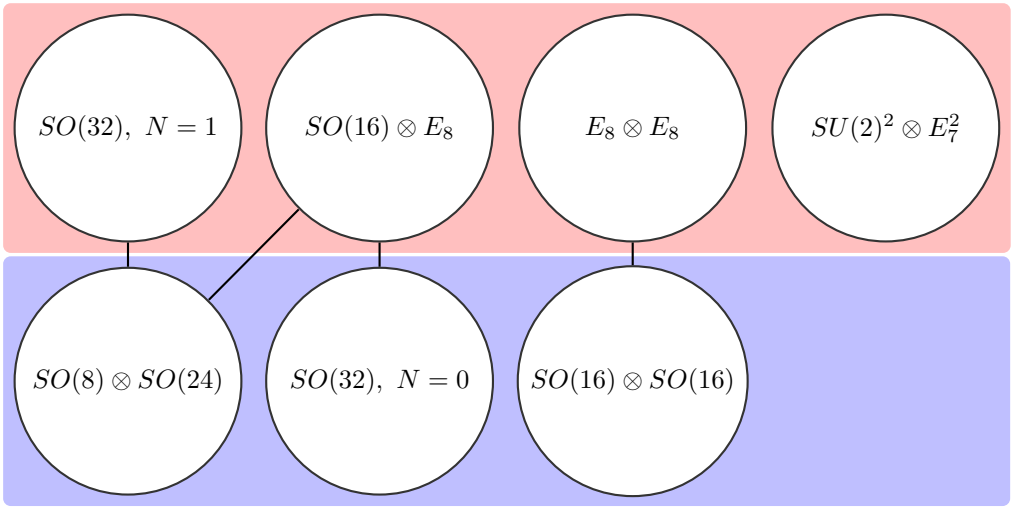


Figure 4.3: A schematic showing the systematic search for two basis vectors of order 2. The columns are models that are produced by different basis vectors, while the rows represent the possible k_{ij} inputs. The lines indicate two models that were produced by the same basis vector set, but different k_{ij} matrices. Therefore, a model with two lines was built by two different sets of basis vectors that produced different models when k_{ij} was changed.

4.3 Combinations with Two Order-2 Layers

Presented in this section are the results of systematic searches with two order-2 basis vectors. In one of these searches, the k_{ij} matrix is varied whilst changing the basis vectors, and in the other it is held fixed.

4.3.1 Varying k_{ij}'s

Figure 4.3 is a graphical representation of the basis vectors and the models they map to. It can be viewed as a "summary" of the systematic search for this order/layer combination. It is important to note that different basis vectors must be used to build all of the models in the space; one cannot simply choose a basis set and only vary the GSO coefficient matrix (henceforth referred to as the k_{ij} matrix) and get all of the models for these layer/order combinations. It would be worthwhile to find a set of basis vectors that produces all of the models by only varying the k_{ij} matrix.

4.3.2 Fixed k_{ij}'s

Fixing the k_{ij} inputs to be equal to $\begin{pmatrix} 1 \\ 1 \\ 1 & 1 \end{pmatrix}$ does not prohibit solutions from being found - all of the models in Figure 4.3 are present. The comparison of the search results between these two methods are presented in Table 4.5. Notice that if the k_{ij} matrix entries for either search have only one value (either all 0's or all 1's) on the final row (the added basis vectors projecting onto the others in the GSOPs), then the basis vectors from both searches match. If the k_{ij} matrix in the search on the left had multiple values for k_{ij} matrix entries, then the corresponding fixed k_{ij} basis vector has more "breaks" - regions with matching boundary conditions. This suggests a relationship between the k_{ij} matrix "breaking" (meaning matching/non-matching values) and basis vector "breaking."

4.3.3 Relation to Order 4 Basis Vectors

One may naively think that a single order-4 basis vector will be able to mimic the degrees of freedom available to two order-2 basis vectors, since $2 \times 2 = 4$. However, closer examination makes it clear that the two data sets have different degrees of freedom. These degrees of freedom are best considered in the context of possible regions of matching boundary conditions, which determine the complexity of the model[5] . Consider first a single basis vector of order 2. Since there are two values available for the basis vectors to acquire and, since all the boundary conditions for the all-periodic basis vector are identical, reordering of the elements (a redundancy for fermion modes that have matching boundary conditions in the other basis vectors in the model) gives two regions of matching boundary conditions.

$$(\vec{1}^{\,8} \quad || \quad (0,...,0) \quad (1,...,1)) \tag{4.35}$$

[5] In some cases there are enhancements to the gauge symmetries, so some higher layer/order models will actually produce fewer gauge groups of larger rank. These instances tend to be rare.

Table 4.5: Comparison of the results between the searches in which the k_{ij} matrix was and was not varied for two order-2 basis vectors. The inputs on the left were generated with multiple k_{ij}'s, while the inputs on the right were generated with a fixed k_{ij}.

O2 O2, varying k_{ij}		Model	O2, O2, fixed k_{ij}	
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SO(32),\ N=1$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 1 \end{pmatrix}$	$SO(8)\otimes SO(24)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,20}\ \vec{1}^{\,4}\ \vec{0}^{\,4}\ \vec{1}^{\,4})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$E_8\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 1 \end{pmatrix}$	$SO(16)\otimes SO(16)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,8}\vec{0}^{\,8})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SO(16)\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\ \vec{1}^{\,12}\ \vec{0}^{\,4}\ \vec{1}^{\,4})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SU(2)^2\otimes E_7^2$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\ \vec{1}^{\,12}\ \vec{0}^{\,4}\ \vec{1}^{\,4})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,8}\ \vec{1}^{\,24})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 1 \end{pmatrix}$	$SO(32),\ N=0$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,8}\ \vec{1}^{\,24})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,4}\ \vec{1}^{\,4}\ \vec{0}^{\,4}\ \vec{1}^{\,20})$	$\begin{pmatrix} 1 \\ 1 \\ 1\ 1 \end{pmatrix}$

Adding a second order-2 basis vector splits the regions up into four sets of matching boundary conditions.

$$\begin{array}{l} (\vec{1}^{\,8} \quad || \quad (0,.....,0) \quad (1,.....,1) \quad) \\ (\vec{1}^{\,8} \quad || \quad (0,...,0) \quad (1,...,1) \quad (0,...,0) \quad (1,...,1)) \end{array} \tag{4.36}$$

For an order-4 basis vector, one assumes that there are still four regions of matching boundary conditions, $(0, \frac{1}{2}, 1, -\frac{1}{2})$. However, because the order-4 basis vector is the first layer with complex (neither integer nor half-integer) phases, there is a symmetry regarding the sign of the those phases. Effectively,

$$-\frac{1}{2} \approx \frac{1}{2} \tag{4.37}$$

for all complex phases in the basis vector. Thus, the regions of matching boundary conditions for a single order-4 basis vector are

$$(\vec{1}^{\,8} \quad || \quad (0,...,0) \quad (\tfrac{1}{2},...,\tfrac{1}{2}) \quad (1,...,1)) \tag{4.38}$$

Additionally, the presence of complex phases add sectors to the model that produce fractional charge elements. Order-2 basis vectors produce only half-integer elements in the charge vectors coming from twisted sectors, while an order-4 basis vector produces not only the half integer twisted states, but also quarter integer twisted states. These additional degrees of freedom granted to the state vectors may result in additional symmetries that are unavailable to the sets of order-2 basis vectors. Comparisons between these data sets have been tabulated in Table 4.6.

Note that the order-4 data set has the $SU(16) \otimes U(1)$ model, while it lacks the $E_8 \otimes E_8$ model. Untwisted boson sectors produce $SO(32)$ gauge groups in ten large space-time dimensions. These are broken by the GSO projections of the twisted sectors into smaller $SO(2n)$ and $SU(n)$ groups. To produce $SU(16)$, a rank 15 gauge group, the twisted sectors must span the entire group charge space. In addition, the twisted sectors must be independent - that is, the basis vectors that produced them

Table 4.6: This table contains each model with the respective pair of order-2 and order-4 basis vectors, as well as the corresponding k_{ij}'s. Note that not all of the models can be produced from each data set. The basis vectors in this table are presented in a real basis.

O2O2		Model	O4	
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SO(32),\ N=1$	$(\vec{1}^{\,8}\|\|(\frac{\vec{2}}{4})^{32})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SO(8)\otimes SO(24)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,22}(\frac{\vec{2}}{4})^{8}\vec{1}^{\,2})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$E_8\otimes E_8$	N/A	N/A
$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 1 \end{pmatrix}$	$SO(16)\otimes SO(16)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,16}(\frac{\vec{2}}{4})^{16})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,16})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SO(16)\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{2}}{4})^{8}\vec{1}^{\,10})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\vec{1}^{\,12}\vec{0}^{\,4}\vec{1}^{\,4})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 0 \end{pmatrix}$	$SU(2)^2\otimes E_7^2$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,12}(\frac{\vec{2}}{4})^{16}\vec{1}^{\,4})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,16})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,8}\vec{1}^{\,24})$	$\begin{pmatrix} 1 \\ 0 \\ 0\ 1 \end{pmatrix}$	$SO(32),\ N=0$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,6}(\frac{\vec{2}}{4})^{8}\vec{1}^{\,18})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$
N/A	N/A	$SU(16)\otimes U(1)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,6}(\frac{\vec{2}}{4})^{24}\vec{1}^{\,2})$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$

must be different - but not orthogonal. This independence allows for contributions from one twisted sector to remove a root coming from the untwisted sector from the possible simple roots of the gauge group, making it an $SU(n) \otimes U(1)$ group rather than an $SO(2n)$ group. A pair of order 2 basis vectors does not produce enough independent twisted sectors to do this. The order 4 basis vector puts the weights of the adjoint representation in a twisted basis with charges of $\frac{1}{4}$, $-\frac{1}{4}$, $\frac{3}{4}$, $-\frac{3}{4}$ in addition to the half-integer charges, so three independent twisted sectors are not needed if there are more than half integer charges present in the model.

By contrast, the $E_8 \otimes E_8$ model does require the twisted sectors to be orthogonal and independent, which cannot be done with a single order-4 basis vector. In an odd ordered model, the lack of half integer twists produces orthogonal sectors in a twisted basis, allowing the $E_8 \otimes E_8$ model to appear for those, but not for order-4 models.

To summarize, the product of the orders does not determine the model spectrum - all possible combinations of orders must be investigated to fully map the model space produced by the free fermionic heterotic construction.

4.4 The Full D=10, Level-1 Heterotic Landscape

The full spectrum of $D = 10$, level-1 models, as mentioned in section 4.1, consists of the eight models presented in table 4.1. The lowest order for which all the models were built out of a single basis vector was order 6. It is worth testing whether one order-3 and one order-2 basis vector will also produce the full range of models, since $2 \times 3 = 6$. While similar reasoning was not true for order-4 and two order-2's, this is indeed the case here. The models, and the basis vectors that produced them, are tabulated in Table 4.7. Schematically, the O3O2 search is summarized in Figure 4.4.

Table 4.7: The models along with the inputs from each data set that produced that model. The basis vectors are presented in a real basis. Note that some of the order-6 basis vectors are actually order-3. Specifically, the $SO(32),\ N=1$ model is produced by the same order-3 basis vectors. The additional order-2 basis vector's contribution is completely projected out.

O6		Model	O3O2	
$(\vec{1}^{\,8}\|\|\vec{0}^{\,26}(\frac{\vec{2}}{3})^6)$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SO(32),\ N=1$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,26}(\frac{\vec{2}}{3})^6)$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\left(\begin{smallmatrix}1&\\1&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}(\frac{\vec{1}}{3})^6\ \vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SO(8)\otimes SO(24)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,26}(\frac{\vec{2}}{3})^6)$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\left(\begin{smallmatrix}1&\\0&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{1}}{3})^{12}\vec{1}^{\,4})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SO(16)\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{2}}{3})^{18})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\ \vec{1}^{\,8})$	$\left(\begin{smallmatrix}1&\\0&\\0&\frac{2}{3}\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{1}}{3})^{16}(\frac{\vec{2}}{3})^2)$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SO(16)\otimes SO(16)$	$(\vec{1}^{\,8}\|\|\vec{0}^{26}(\frac{\vec{2}}{3})^6)$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\left(\begin{smallmatrix}1&\\0&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{2}}{3})^{18})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$E_8\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,26}(\frac{\vec{2}}{3})^6)$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\ \vec{1}^{\,16})$	$\left(\begin{smallmatrix}1&\\1&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,12}(\frac{\vec{1}}{3})^{14}(\frac{\vec{2}}{3})^4\vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SU(2)^2\otimes E_7^2$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{2}}{3})^{18})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\vec{1}^{\,2}\ \vec{0}^{\,12}\ \vec{1}^{\,6})$	$\left(\begin{smallmatrix}1&\\0&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|(\frac{\vec{1}}{3})^{22}(\frac{\vec{2}}{3})^2\ \vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SU(16)\otimes U(1)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,8}(\frac{\vec{2}}{3})^{\,24})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,6}\vec{1}^{\,2}\ \vec{0}^{\,18}\ \vec{1}^{\,6})$	$\left(\begin{smallmatrix}1&\\1&\\0&0\end{smallmatrix}\right)$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,8}(\frac{\vec{1}}{3})^{\,12}\ \vec{1}^{\,12})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$	$SO(32),\ N=0$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,8}(\frac{\vec{2}}{3})^{24})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,8}\ \vec{1}^{\,24})$	$\left(\begin{smallmatrix}1&\\1&\\1&0\end{smallmatrix}\right)$

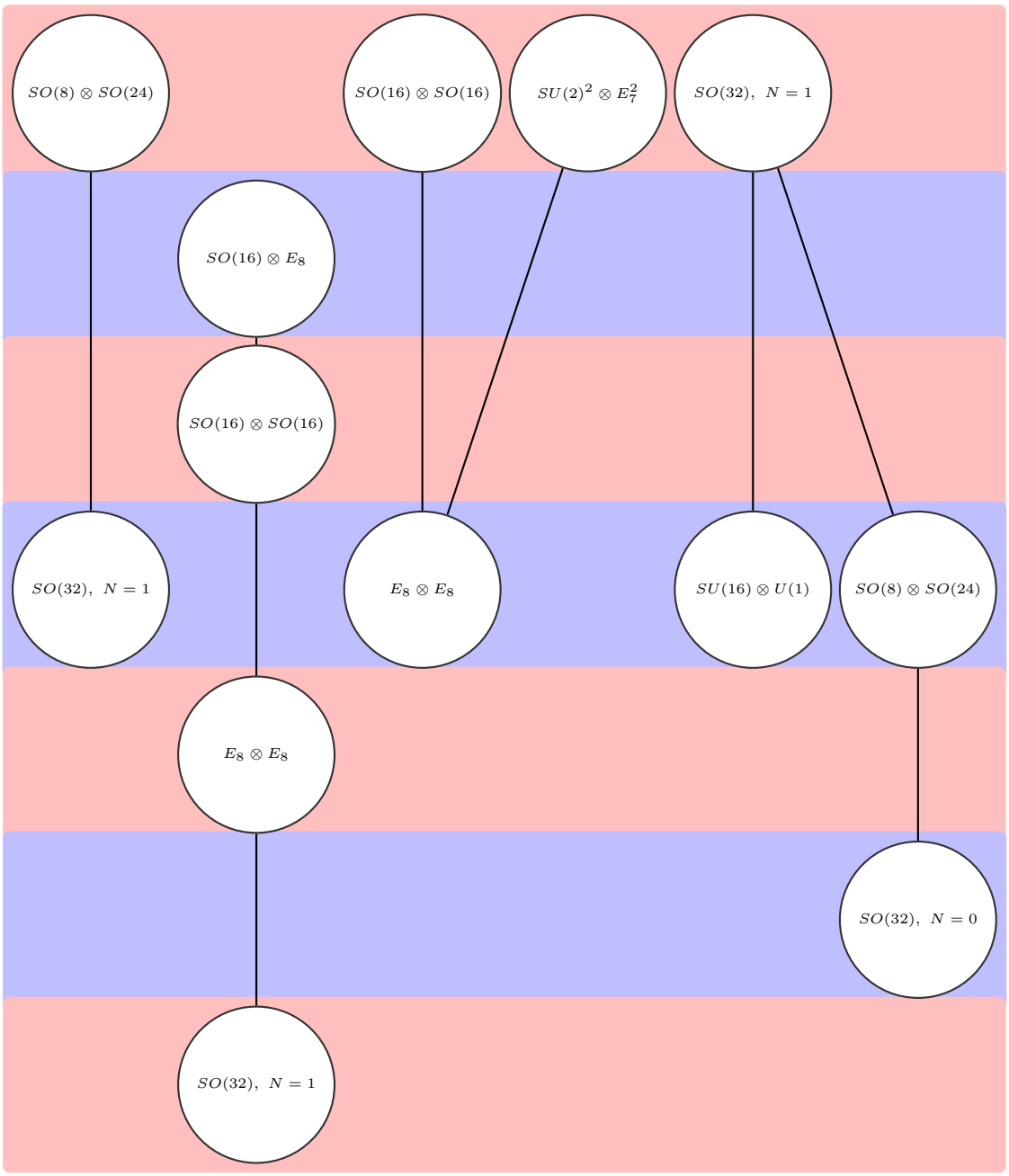


Figure 4.4: A schematic diagram of the O3O2 systematic search. The different columns represent different basis vectors, while the different rows represent possible k_{ij} matrix configurations. Lines connect models produced by the same basis vector, but different k_{ij} matrices.

Also of interest is whether the full $D = 10$, level-1 model spectrum can be produced using only periodic/anti-periodic modes (order-2 basis vectors). It can indeed, but only with sets of three basis vectors. They are tabulated adjacent to their order-6 counterparts in table 4.8. Schematically, the search is summarized in figure 4.5.

4.5 Conclusions

To conclude, we decided to examine the $D = 10$, level-1 heterotic models to deduce redundancies in the WCFFHS construction method. We conjectured and proved that for all models with a single basis vector, odd ordered right mover and massless left mover, the maximum number of ST SUSYs were present. This implies that searches of this sort in lower space-time dimensions will contain either models with the maximum number of space-time SUSYs or models without space-time supersymmetry. Specifically, in ten dimensions this means single basis vector searches with odd ordered right movers will only have two models: $SO(32)$ and $E_8 \otimes E_8$, both with $N = 1$ space-time supersymmetry.

For searches of two basis vector models, we showed that the basis vectors must be varied to fully map out the model spectrum. The GSO coefficient matrix on the other hand, does not necessarily need to be varied if the basis vectors can produce enough sets of matching boundary conditions. We also showed that the product of the orders across which the search is performed does not necessarily dictate the model spectrum. In particular, we showed that all modular invariant combinations of two order-2 basis vectors do not produce the same models as all possible order-4 basis vectors.

Finally, we showed that the lowest order for which all $D = 10$, level-1 models could be produced from a single basis vector is 6. The lowest combination of orders that produces all of the above mentioned from pairs of basis vectors is O3O2. The

Table 4.8: The models along with the inputs from each data set producing that model. The basis vectors are expressed in a real basis.

O6	Model	O2O2O2	
$(\vec{1}^{\,8}\|\|\vec{0}^{\,26}(\frac{\vec{2}}{3})^6)$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SO(32),\ N=1$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,20}\vec{1}^{\,4}\vec{0}^{\,4}\vec{1}^{\,4})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 0\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}(\frac{\vec{1}}{3})^6\vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SO(8)\otimes SO(24)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,20}\vec{1}^{\,4}\vec{0}^{\,4}\vec{1}^{\,4})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 1\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,16}(\frac{\vec{1}}{3})^{12}\vec{1}^{\,4})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SO(16)\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,8}\vec{0}^{\,8})$	$\begin{pmatrix}1\\0\\0\ 1\\0\ 1\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{1}}{3})^{16}(\frac{\vec{2}}{3})^2)$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SO(16)\otimes SO(16)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,8}\vec{0}^{\,8})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 1\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,14}(\frac{\vec{2}}{3})^{18})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $E_8\otimes E_8$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,8}\vec{0}^{\,8})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 0\ 1\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,12}(\frac{\vec{1}}{3})^{14}(\frac{\vec{2}}{3})^4\vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SU(2)^2\otimes E_7^2$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,32})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\vec{1}^{\,12}\vec{0}^{\,4}\vec{1}^{\,4})$	$\begin{pmatrix}1\\0\\0\ 1\\0\ 0\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,6}(\frac{\vec{1}}{3})^{22}(\frac{\vec{2}}{3})^2\vec{1}^{\,2})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SU(16)\otimes U(1)$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,12}\vec{1}^{\,12}\vec{0}^{\,4}\vec{1}^{\,4})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,6}\vec{1}^{\,6}\vec{0}^{\,6}\vec{1}^{\,6}\vec{0}^{\,2}\vec{1}^{\,2}\vec{0}^{\,2}\vec{1}^{\,2})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 0\ 0\end{pmatrix}$
$(\vec{1}^{\,8}\|\|\vec{0}^{\,8}(\frac{\vec{1}}{3})^{12}\vec{1}^{\,12})$	$\left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ $SO(32),\ N=0$	$(\vec{1}^{\,8}\|\|\vec{0}^{\,24}\vec{1}^{\,8})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,16}\vec{1}^{\,16})$ $(\vec{1}^{\,8}\|\|\vec{0}^{\,8}\vec{1}^{\,8}\vec{0}^{\,8}\vec{1}^{\,8})$	$\begin{pmatrix}1\\0\\0\ 0\\0\ 0\ 1\end{pmatrix}$

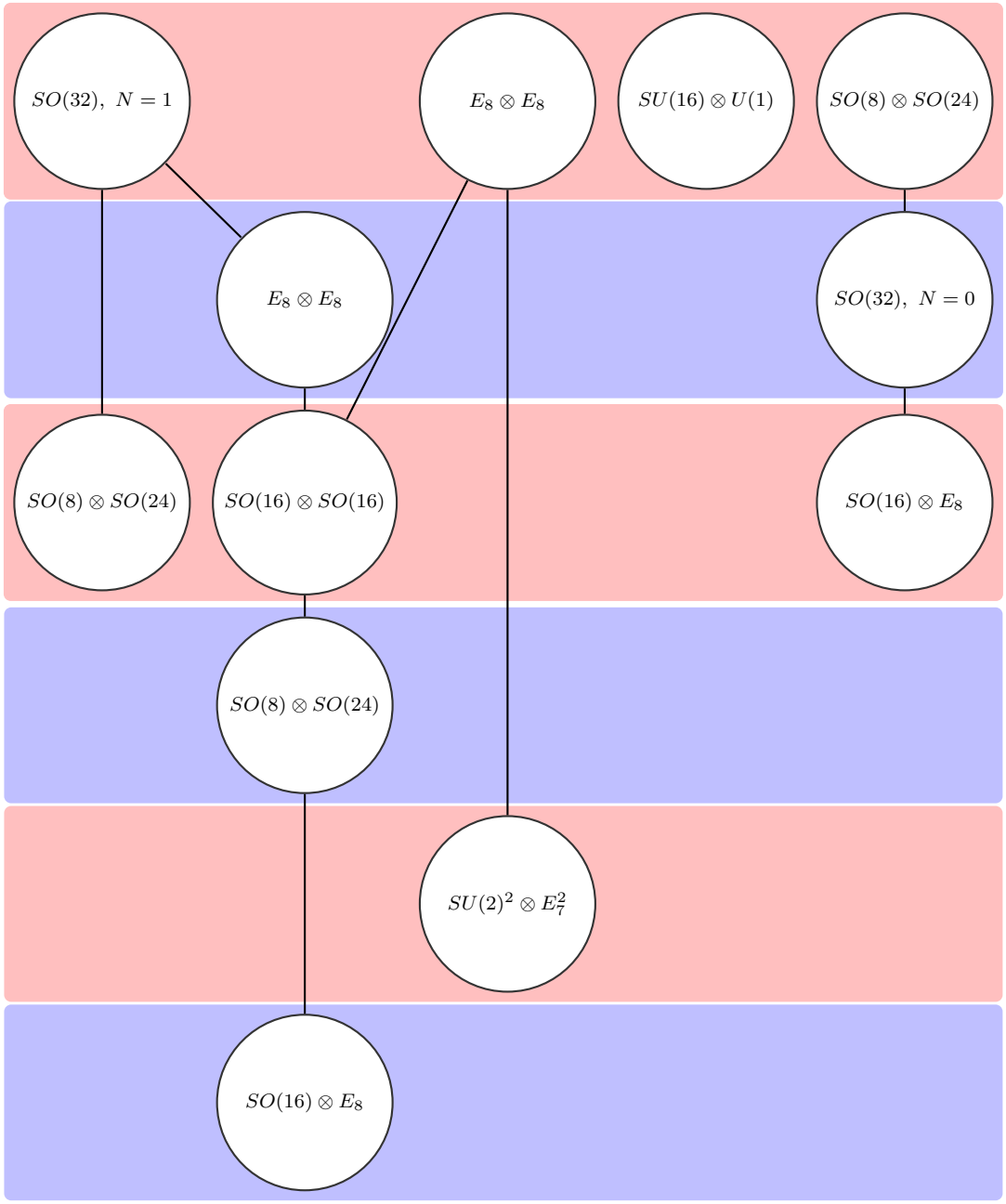


Figure 4.5: A schematic diagram of the O2O2O2 search. As with the other diagrams, the different columns indicate different basis vectors, while different rows represent different k_{ij}'s. Lines indicate models produced by identical basis vectors, but different k_{ij} matrices.

lowest number of order-2 basis vectors needed to produce all D=10, level-1 models is three.

The ultimate conclusion of this study is that for simple models, it is very possible to correlate the basis vector and GSO coefficient inputs with the particle content output to a certain extent. These correlations make it possible to further narrow searches by analytically and statistically isolating the properties of basis vectors that give phenomenologically realistic results. Additional work in this area will be to explicitly map out D=8 and D=6 heterotic string models, searching for extra redundancies that occur when there are compact space-time dimensions.

CHAPTER FIVE

Preliminary Systematic NAHE Investigations

This chapter will outline the results of a few single layer extensions to the NAHE set of basis vectors. The NAHE set consists of five order-2 basis vectors which have phenomenology conducive to realistic heterotic string models. First, this set will be discussed in detail, then statistics for systematic order-2 and order-3 extensions will be examined. After that, statistics for models with specific GUT groups will be discussed. Special emphasis will be placed on the number of chiral matter generations present for each GUT group model. Finally, models with three chiral matter generations will be discussed. These are the first three-generation models constructed which have a geometric interpretation.

5.1 The NAHE Set

The NAHE[65] set is a set of five order-2 basis vectors which have served as a common basis set for phenomenologically realistic WCFFHS models. The basis vectors which generate this set are given in Table 5.1. The massless particle spectrum is given in Table 5.2. The NAHE set's particle content, in addition to the particles listed in Table 5.2, contains an N=1 ST SUSY. The observable sector of the NAHE set is an $SO(10)$ GUT group with three sixteen dimensional matter representations serving as the generations of matter. Each generation is charged under a different $SU(4)$ gauge group, and there are two copies of each representation. There are two distinct representations with $SO(10)$ charge 16 and $SU(4)$ charge 4, since the $SU(4)$ charge has a barred and unbarred representation. This brings the total number of copies for each generation up to four. In addition, the dimension of the $SU(4)$ charge itself is counted as being a set of copies of each generation, so the total number of copies of each generation is sixteen. There are no matter representations charged

Table 5.1: The basis vectors and GSO coefficients of the NAHE set arranged into sets of matching boundary conditions. N_R is the order of the right mover. The elements ψ, $\overline{\psi}^{\,i}$, $\overline{\eta}^{\,i}$, and $\overline{\phi}^{\,i}$ are expressed in a complex basis, while x^i, y^i, w^i, $\overline{y}^{\,i}$, and $\overline{w}^{\,i}$ are expressed in a real basis.

Sec	N_R	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{\,1,\ldots,5}$	$\overline{\eta}^{\,1}$	$\overline{\eta}^{\,2}$	$\overline{\eta}^{\,3}$	$\overline{\phi}^{\,1,\ldots,8}$
$\vec{1\!\!1}$	2	1	1	1	1	1,...,1	1	1	1	1,...,1
$\vec{S}$	2	1	1	1	1	0,...,0	0	0	0	0,...,0
$\vec{b}_1$	2	1	1	0	0	1,...,1	1	0	0	0,...,0
$\vec{b}_2$	2	1	0	1	0	1,...,1	0	1	0	0,...,0
$\vec{b}_3$	2	1	0	0	1	1,...,1	0	0	1	0,...,0

Sec	N_R	$y^{\,1,2}w^{\,5,6}\|\|\overline{y}^{\,1,2}\overline{w}^{\,5,6}$	$y^{\,3,\ldots,6}\|\|\overline{y}^{\,3,\ldots,6}$	$w^{\,1,\ldots,4}\|\|\overline{w}^{\,1,\ldots,4}$
$\vec{1\!\!1}$	2	1,...,1 || 1,...,1	1,...,1 || 1,...,1	1,...,1 || 1,...,1
$\vec{S}$	2	0,...,0 || 0,...,0	0,...,0 || 0,...,0	0,...,0 || 0,...,0
$\vec{b}_1$	2	0,...,0 || 0,...,0	1,...,1 || 1,...,1	0,...,0 || 0,...,0
$\vec{b}_2$	2	1,...,1 || 1,...,1	0,...,0 || 0,...,0	0,...,0 || 0,...,0
$\vec{b}_3$	2	0,...,0 || 0,...,0	0,...,0 || 0.,,,.0	1,...,1 || 1,...,1

$$k_{ij} = \left(\begin{array}{c|ccccc} & \vec{1\!\!1} & \vec{S} & \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \\ \hline \vec{1\!\!1} & 1 & 0 & 1 & 1 & 1 \\ \vec{S} & 0 & 0 & 0 & 0 & 0 \\ \vec{b}_1 & 1 & 1 & 1 & 1 & 1 \\ \vec{b}_2 & 1 & 1 & 1 & 1 & 1 \\ \vec{b}_3 & 1 & 1 & 1 & 1 & 1 \end{array}\right)$$

Table 5.2: The particle content of the model produced by the NAHE set.

QTY	$SU(4)$	$SU(4)$	$SU(4)$	$SO(10)$	E_8
2	$\overline{4}$	1	1	16	1
2	1	$\overline{4}$	1	16	1
2	1	1	$\overline{4}$	16	1
2	1	1	4	16	1
1	1	1	6	10	1
2	1	4	1	16	1
1	1	6	1	10	1
1	1	6	6	1	1
2	4	1	1	16	1
1	6	1	1	10	1
1	6	1	6	1	1
1	6	6	1	1	1

under the E_8 group, so it is the designated hidden sector for this model. Extensions of the NAHE set do not necessarily keep the designated observable sectors in the model. A model could, rather than break down the $SO(10)$ gauge group, break the E_8 into an E_6, producing an E_6 observable sector. In the past, when models were constructed individually, this was not common. Most individually constructed NAHE based models broke the $SO(10)$ gauge group into a Pati-Salam ($SU(4) \otimes SU(2) \otimes SU(2)$) group, $SU(5) \otimes U(1)$, or the MSSM gauge group ($SU(3) \otimes SU(2) \otimes U(1)$). The present study takes a different approach; to start with the NAHE basis and build all possible basis vectors with modular invariance for a given order and layer, then examine the models statistically. The notation for the next sections will change slightly; from here on the term layer will refer to the number of basis vectors after the initial set of five NAHE vectors rather than the total number of basis vectors.

5.2 Statistics for Order-2 Layer-1

The first set of statistics to be reported here are for extensions to the NAHE set with a single basis vector of order 2. The GSO coefficients were fixed for the NAHE set to those presented in Table 5.1. The GSO coefficients for the extended basis vector were systematically generated such that all possible combinations consistent with modular invariance were built. This study was repeated for the NAHE set without the ST SUSY generating basis vector $\vec{S}$ to determine its effect on the models produced.

5.2.1 With $\vec{S}$

There were 439 unique models produced out of 1,945,088 total consistent models. Approximately 9.5% of the models in the data set without rank-cuts were duplicates, and 13% of the models with rank-cuts were duplicates. All duplicates were removed prior to the statistical analysis. The frequency of the individual groups appearing in the unique models is presented in Table 5.3. The first item of note is how many models retain at least one of the original gauge groups from the NAHE set. Approximately 77% of the models kept at least one $SU(4)$ gauge group, while $\approx$ 36% of the models kept their $SO(10)$ and $\approx$ 33% kept their E_8. The most common gauge group in this set is $SU(2)$, which is expected. $SU(2)$ is the lowest rank non-Abelian gauge group attainable. About 17% have an $SU(2)^{(2)}$ gauge group. In these models, this happens when a left moving mode is paired with a right moving mode. As mentioned earlier, left-right paired elements reduce the rank of the gauge lattice of the model. Hence, left-right pairs are referred to as rank cuts. The non-simply laced gauge group $SO(5)$ also appears due to rank cuts.

Also of interest is the number of models with a $U(1)$ gauge group, which for this data set is quite high. $U(1)$'s can be problematic when dealing with deeper phenomenology in a model. The more $U(1)$'s present in a model the more likely

Table 5.3: The frequency of the individual gauge groups amongst the unique models for the NAHE + O2L1 data set. Gauge groups at Kač-Moody level higher than 1 are denoted with a superscript indicating the Kač-Moody level.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	365	83.14%
$SU(2)^{(2)}$	73	16.63%
$SU(4)$	338	76.99%
$SU(6)$	2	0.4556%
$SU(8)$	2	0.4556%
$SO(5)$	155	35.31%
$SO(8)$	141	32.12%
$SO(10)$	160	36.45%
$SO(12)$	2	0.4556%
$SO(14)$	3	0.6834%
$SO(16)$	147	33.49%
$SO(18)$	1	0.2278%
$SO(20)$	2	0.4556%
$SO(22)$	1	0.2278%
$SO(24)$	1	0.2278%
$SO(26)$	1	0.2278%
E_6	1	0.2278%
E_7	142	32.35%
E_8	144	32.8%
$U(1)$	332	75.63%

the model is to have anomalous charge. This anomalous charge must be dealt with by finding D- and F-flat directions in the superpotential. However, the more $U(1)$ charges present, the more flat directions a model is likely to have, enabling more flexibility when giving mass to observable sector charged exotics. As that particular process is computationally intensive and has not been fully automated, discussion of anomalous $U(1)$ charges and flat directions is not present in this study.

Though most of the models have smaller individual gauge group components, some models have gauge group enhancements. There are some models with $SO(18)$, $SO(20)$, $SO(22)$, $SO(24)$, and $SO(26)$ gauge groups. Those groups have rank 9, 10, 11, 12, and 13, respectively, making them higher rank than any one of the NAHE set gauge groups. This occurs when an added basis vector bridges the gap between the mutually orthogonal sets of states of the original five basis vectors, unifying the root spaces of the individual groups into one larger group. For order-2 models, however, it is clear this is not common.

Another way of measuring the enhancements that occur is looking at the number of gauge group factors in each model. Those are plotted in Figure 5.1. There is a definite peak at nine gauge group factors, much higher than the five initial gauge groups present in the NAHE set. Note that there are not many models with fewer than five gauge group factors. This implies that though there are several enhanced groups of higher rank than the initial NAHE set, the other gauge groups in the model remain broken.

Relevant GUT groups and the number of unique models containing those groups is presented in Table 5.4. The relatively low number of $SU(n+1)$ groups (excluding $SU(4)$) explains the lack of GUT group models in this data set.

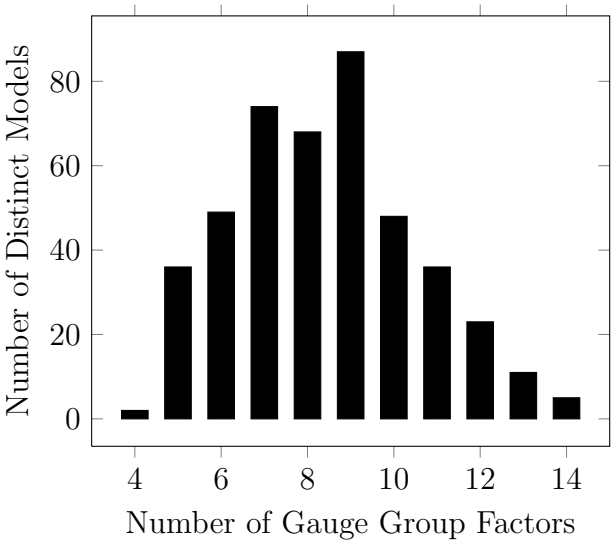


Figure 5.1: The number of gauge group factors for each model in the NAHE + O2L1 data set.

Table 5.4: The number of unique models containing GUT groups for the NAHE + O2L1 data set.

GUT Group	Number of Unique Models	% of Unique Models
E_6	1	0.2278%
$SO(10)$	160	36.45%
$SU(5) \otimes U(1)$	0	0%
$SU(4) \otimes SU(2) \otimes SU(2)$	243	55.35%
$SU(3) \otimes SU(2) \otimes SU(2)$	0	0%
$SU(3) \otimes SU(2) \otimes U(1)$	0	0%

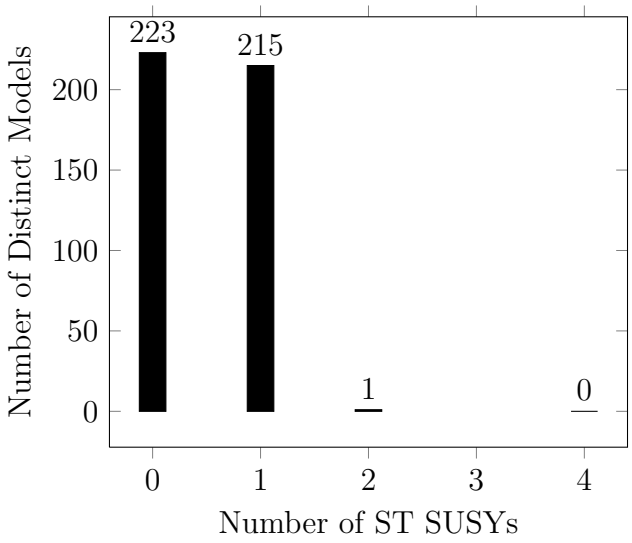


Figure 5.2: The number of ST SUSYs for the NAHE + O2L1 data set.

Table 5.5: A basis vector and k_{ij} matrix row which produces an enhanced ST SUSY when added the NAHE set.

Sec	O	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,\ldots,5}$	$\overline{\eta}^{1}$	$\overline{\eta}^{2}$	$\overline{\eta}^{3}$	$\overline{\phi}^{1,\ldots,8}$
$\vec{v}$	2	1	1	1	1	0,...,0	1	1	0	0,...,0

Sec	O	$y^{1,2}w^{5,6}\|\|\overline{y}^{1,2}\overline{w}^{5,6}$	$y^{3,\ldots,6}\|\|\overline{y}^{3,\ldots,6}$	$w^{1,\ldots,4}\|\|\overline{w}^{1,\ldots,4}$
$\vec{v}$	2	0,0,1,1||1,1,1,1	0,0,1,1||1,1,1,1	0,...,0||0,...,0

$$k_{\vec{v},j} = (0,\ 0,\ 0,\ 1,\ 1)$$

The number of ST SUSYs are plotted against the number of unique models in Figure 5.2. Many of the basis vector extensions did not alter the ST SUSY, while about half reduced it. More interestingly, there is one model with enhanced ST SUSY. The basis vector for that model is presented in Table 5.5. The particle content of this model is presented in Table 5.6. The gauge groups of this model are identical to those of the NAHE set, but with fewer matter representations, particularly with regard to the $SU(4)$ charges. The enhanced ST SUSY comes from a new gravitino generating sector $\vec{b}_1 + \vec{b}_2 + \vec{v}$, which contributes a single gravitino

Table 5.6: The particle content of the $N = 2$ ST SUSY NAHE based model.

QTY	$SU(4)$	$SU(4)$	$SU(4)$	$SO(10)$	E_8
1	1	$\overline{4}$	1	$\overline{16}$	1
1	1	$\overline{4}$	1	16	1
1	1	1	$\overline{4}$	$\overline{16}$	1
1	1	1	$\overline{4}$	16	1
1	1	1	4	$\overline{16}$	1
1	1	1	4	16	1
1	1	4	1	$\overline{16}$	1
1	1	4	1	16	1
2	1	6	6	1	1
2	6	1	1	10	1

state to the model. The other gravitino comes from $\vec{S}$. This example highlights the importance of systematic searches; the enhanced ST SUSYs come from very specific basis vectors that combine with the NAHE set to provide unexpected phenomenology. Though there are not a statistically significant number of models with this property, these models can highlight subtleties in the WCFFHS formulation that may go unnoticed in a random search.

The number of $U(1)$ gauge groups are plotted against the number of unique models in Figure 5.3. The greater number of $U(1)$ factors present in the model the greater that model's capacity for carrying anomalous charge. There are relatively few $U(1)$'s in the models of this class. This is likely the result of the basis vectors having only periodic phases — nonzero, non-periodic phases break $SO(2n)$ groups into $SU(n-1) \otimes U(1)$ groups in most cases. Thus, there are not many $U(1)$ groups expected or found in this data set.

Another phenomenological property which might have statistical significance is the number of non-Abelian singlets, as non-Abelian singlets often carry observable sector hypercharge. However, no such particle with this property has yet been

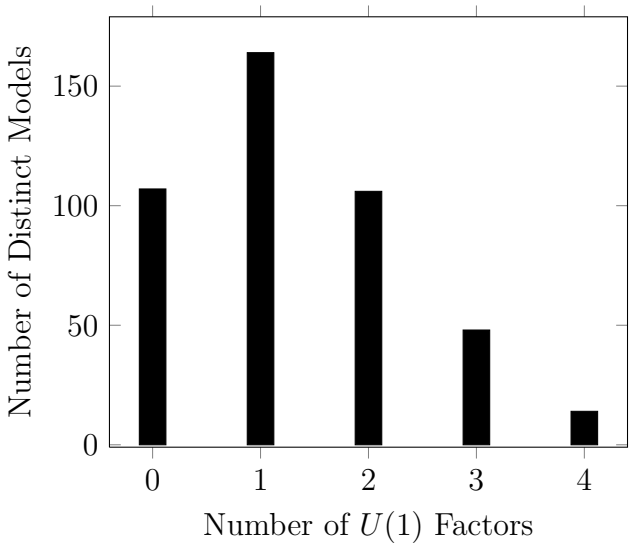


Figure 5.3: The number of $U(1)$ factors for the NAHE + O2L1 data set.

observed. Additionally, singlet particles contribute to the mass-energy density of the model. Too many non-Abelian singlets could result in a mass-energy density higher than observed values, also producing bad phenomenology. The number of non-Abelian singlets for this data set is plotted in Figure 5.2.1

5.2.2 Without $\vec{S}$

Also of interest is the effect of the $\vec{S}$ vector in the set of NAHE basis vectors. Not only does the $\vec{S}$ vector generate ST SUSY, it also adds a degree of freedom to the k_{ij} matrix. It stands to reason that removing $\vec{S}$ will also have an effect on the massless gauge and matter content in addition to the supersymmetry. There are 282 unique models in the set of 1,940,352 consistent models, in contrast to the 439 models in the data set with $\vec{S}$. About 8.4% of the models without rank cuts had duplicates, while 9.4% of the models with rank cuts were duplicates of other models produced. All duplicates were removed from the statistics to follow. Moreover the 282 models in this set do not all belong to the $N = 0$ models in Figure 5.2. In

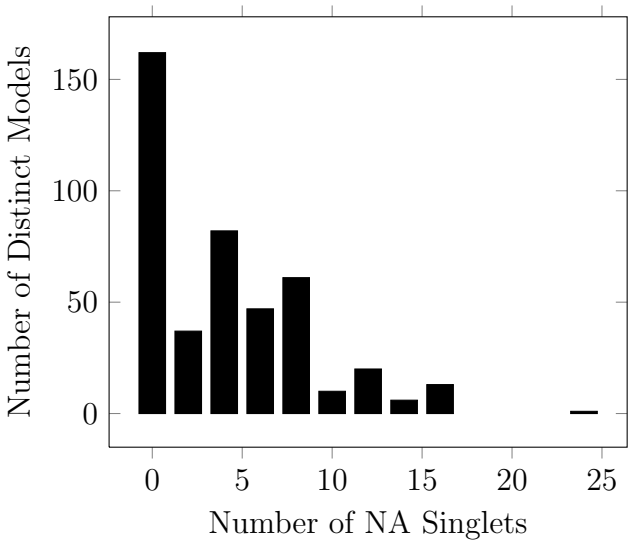


Figure 5.4: The number of non-Abelian singlets in the NAHE + O2L1 data set.

fact, the two data sets have only 8 models in common, implying that the presence or non-presence of $\vec{S}$ has an effect on more than the ST SUSY.

The gauge content of the order-2 layer-1 NAHE extensions without the $\vec{S}$ vector are presented in Table 5.7. A side-by-side comparison of the gauge group content percentages is shown in Table 5.8. The similarities between the gauge content of these two data sets are striking. It is tempting to assume that ignoring ST SUSYs when determining uniqueness will result in the data sets being identical, since the ST SUSY generator $\vec{S}$ is the only real difference between the data sets. This is not the case. In fact, ignoring ST SUSYs when comparing the intersection of these two sets of models gives the same number of models common to both sets: 8. This implies that the matter representations are affected by whether $\vec{S}$ is in the set of basis vectors making up a model. What is likely occurring in the models with $\vec{S}$ is that the sector coming from $\vec{S}$ is contributing non-adjoint representations to the matter states of the model. In other words, gravitinos are not the only fermion states coming from $\vec{S}$. In order to begin fully mapping the heterotic landscape the full effect of leaving out the $\vec{S}$ vector should be examined, as its presence may affect

Table 5.7: The gauge content of the NAHE + O2L1 data set without $\vec{S}$.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	233	82.62%
$SU(2)^{(2)}$	67	23.76%
$SU(4)$	212	75.18%
$SU(6)$	2	0.7092%
$SU(8)$	2	0.7092%
$SO(5)$	107	37.94%
$SO(8)$	89	31.56%
$SO(10)$	100	35.46%
$SO(12)$	2	0.7092%
$SO(14)$	3	1.064%
$SO(16)$	83	29.43%
$SO(18)$	1	0.3546%
$SO(20)$	2	0.7092%
$SO(22)$	1	0.3546%
$SO(24)$	1	0.3546%
$SO(26)$	1	0.3546%
E_6	1	0.3546%
E_7	95	33.69%
E_8	98	34.75%
$U(1)$	209	74.11%

Table 5.8: A side-by-side comparison of the gauge content for NAHE + O2L1 with and without $\vec{S}$.

Gauge Group	With $\vec{S}$	Without $\vec{S}$
$SU(2)$	83.14%	82.62%
$SU(2)^{(2)}$	16.63%	23.76%
$SU(4)$	76.99%	75.18%
$SU(6)$	0.4556%	0.7092%
$SU(8)$	0.4556%	0.7092%
$SO(5)$	35.31%	37.94%
$SO(8)$	32.12%	31.56%
$SO(10)$	36.45%	35.46%
$SO(12)$	0.4556%	0.7092%
$SO(14)$	0.6834%	1.064%
$SO(16)$	33.49%	29.43%
$SO(18)$	0.2278%	0.3546%
$SO(20)$	0.4556%	0.7092%
$SO(22)$	0.2278%	0.3546%
$SO(24)$	0.2278%	0.3546%
$SO(26)$	0.2278%	0.3546%
E_6	0.2278%	0.3546%
E_7	32.35%	33.69%
E_8	32.8%	34.75%
$U(1)$	75.63%	74.11%

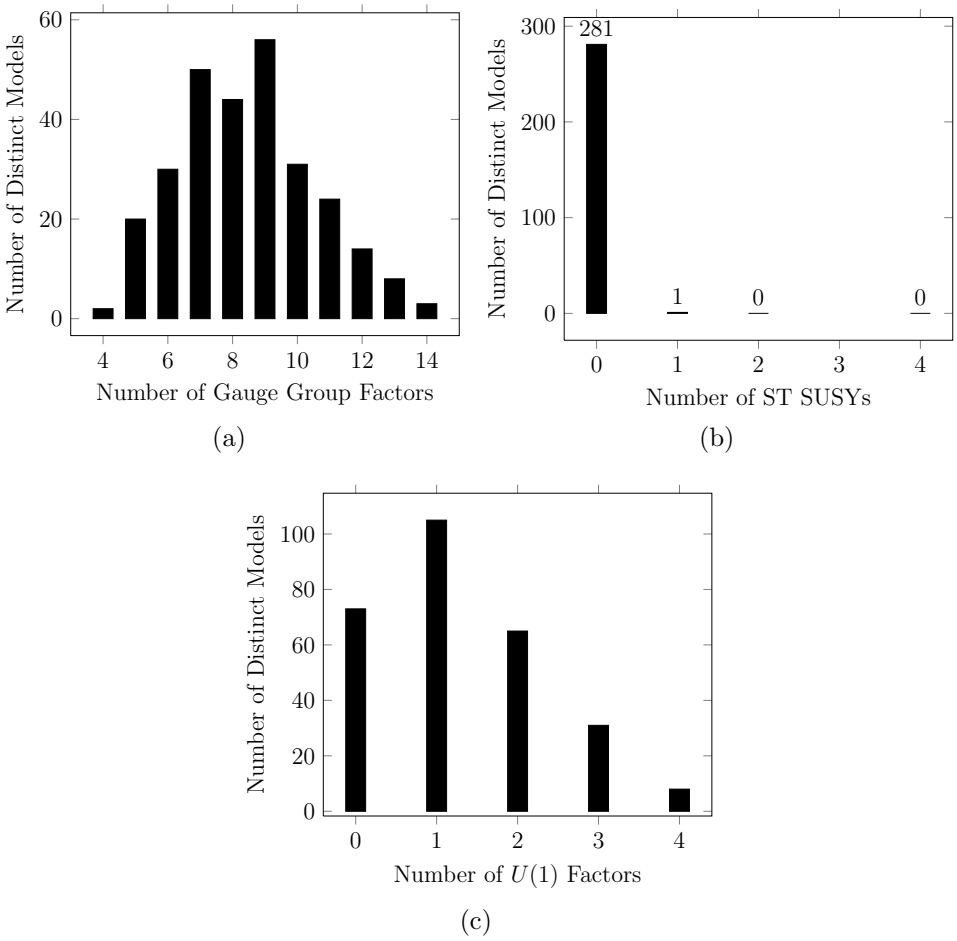


Figure 5.5: Statistics for the NAHE + O2L1 data set without $\vec{S}$.

the likelihood of models with three chiral generations. Other statistics from this data set will now be presented for completeness.

The number of gauge group factors, ST SUSYs, and $U(1)$ factors are plotted in Figure 5.5. None of the models had any non-Abelian singlets, suggesting that the non-Abelian singlets may be coming from the $\vec{S}$ sector. The occurrances of GUT groups are charted in Table 5.9.

5.3 Statistics for Order-3 Layer-1

The next set of statistics to be reported are for extensions to the NAHE set with a single basis vector of right moving order 3. The order 3 basis vectors added

Table 5.9: The number of unique models containing GUT groups for the NAHE + O2L1 data set without $\vec{S}$.

GUT Group	Number of Unique Models	% of Unique Models
E_6	1	0.3546%
$SO(10)$	100	35.46%
$SU(5) \otimes U(1)$	0	0%
$SU(4) \otimes SU(2) \otimes SU(2)$	156	55.32%
$SU(3) \otimes SU(2) \otimes SU(2)$	0	0%
$SU(3) \otimes SU(2) \otimes U(1)$	0	0%

are fermion sectors; the left movers are order-2. Since the orders of the left and right movers are not the same, the total order of the basis vector extensions is 6. The difference between order-3 basis vectors of this type and true order-6 basis vectors is that all six possibilities for the phases will appear in an order-6 right mover, while only three phases appear in an order-3 right mover. The coefficients generating the sectors from these basis vectors still range from 0 to 5, however. This has interesting effects on the fermion spectrum, particularly the gravitinos, as has already been mentioned in the D=10 study. Statistics will be presented for models of this type both with and without $\vec{S}$.

5.3.1 With $\vec{S}$

The presence of $\vec{S}$ in the NAHE set for this search causes any order-3 basis vector with the same left mover as $\vec{S}$ to be inconsistent. This is due to the $\mathbb{Z}_2^L||\mathbb{Z}_3^R$ symmetries of the left and right mover. Adding $\vec{S}$ to three times the basis vector extension results in a second $\vec{0}$ sector, which means the set of basis vectors is not linearly independent. Moreover, since the right mover does not have any periodic

phases, there can be no rank-cutting in these models. Non-simply laced gauge groups and higher level Kač-Moody algebras are not present.

Despite having no linearly independent basis vectors with the same left movers as $\vec{S}$, this data set contains quite a bit more distinct models. There were 373,152 models in this set, but only 3,036 were unique. This relatively (compared to the order-2 extension) high number of unique models suggests that the periodic/anti-periodic phases of the order-2 models have a redundancy not present in models of this type. Based on the double counting of the order-2 models without rank-cuts, the estimated systematic uncertainty for the order-3 statistics is 10%. The gauge content of these models is presented in Table 5.10. The most noticeable difference between Table 5.10 and Table 5.3 is the number of $SU(n+1)$-type gauge groups. The GSO projections of the new sector break the untwisted ($\vec{0}$) sector from $SO(44)$ to smaller $SO(2n)$-type groups. Phases that are neither periodic nor anti-periodic transform these groups from $SO(2n)$ to $SU(n) \otimes U(1)$. Since all of the phases in this set fall into that category, the $SO(2n)$-type groups appear when the states from the added sector are projected out by certain k_{ij} matrix choices. The $SU(n+1)$-type groups are created when the contributions from the added sector are left in the model. The number of gauge group factors per model are plotted in Figure 5.6. As with Figure 5.1, there are still very few models which have less than five gauge group factors. There are also peaks at 10, 11 and 12 gauge group factors, as opposed to just a single peak at 9 in Figure 5.1.

Relevant GUT groups and number of unique models containing those groups are presented in Table 5.11. The number of ST SUSYs for the order-3 data set is plotted in Figure 5.7 Notice that a statistically significant number of models have enhanced ST SUSY. This trend is expected, as every odd-ordered right mover with a massless fermion left mover will produce an additional gravitino generating sector in the model.

Table 5.10: The gauge group content of the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	2587	85.21%
$SU(3)$	923	30.4%
$SU(4)$	2241	73.81%
$SU(5)$	543	17.89%
$SU(6)$	735	24.21%
$SU(7)$	215	7.082%
$SU(8)$	460	15.15%
$SU(9)$	76	2.503%
$SU(10)$	76	2.503%
$SU(11)$	17	0.5599%
$SU(12)$	41	1.35%
$SU(13)$	3	0.09881%
$SU(14)$	5	0.1647%
$SO(8)$	860	28.33%
$SO(10)$	659	21.71%
$SO(12)$	400	13.18%
$SO(14)$	372	12.25%
$SO(16)$	260	8.564%
$SO(18)$	11	0.3623%
$SO(20)$	33	1.087%
$SO(22)$	5	0.1647%
$SO(24)$	15	0.4941%
$SO(26)$	3	0.09881%
E_6	193	6.357%
E_7	147	4.842%
E_8	80	2.635%
$U(1)$	2955	97.33%

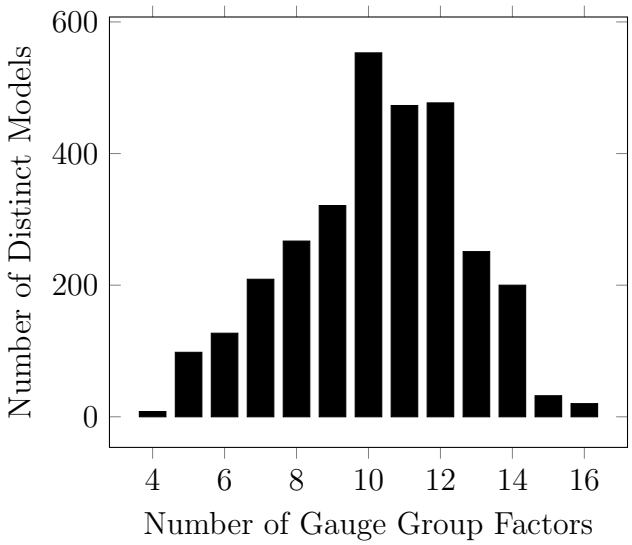


Figure 5.6: The number of gauge group factors per model in the NAHE + O3L1 data set.

Table 5.11: The number of unique models containing GUT groups for the NAHE + O3L1 data set.

GUT Group	Number of Unique Models	% of Unique Models
E_6	193	6.36%
$SO(10)$	659	21.71%
$SU(5) \otimes U(1)$	543	17.89%
$SU(4) \otimes SU(2) \otimes SU(2)$	1648	54.28%
$SU(3) \otimes SU(2) \otimes SU(2)$	628	20.69%
$SU(3) \otimes SU(2) \otimes U(1)$	775	25.53%

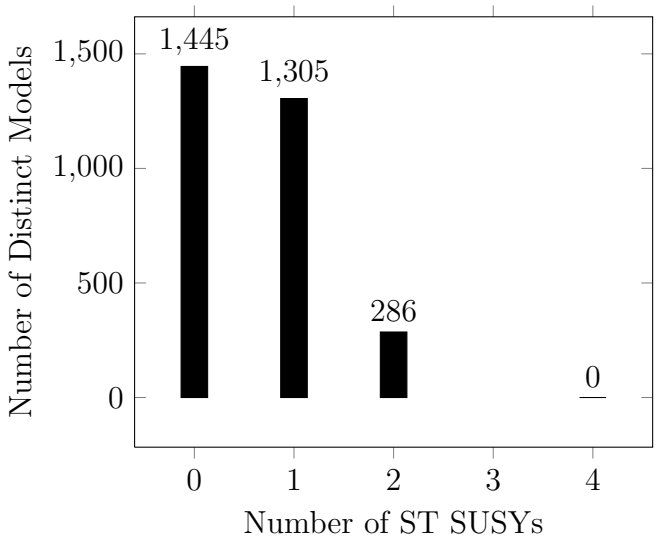


Figure 5.7: The ST SUSYs for the NAHE + O3L1 data set.

Table 5.10 also makes apparent the number of models containing $U(1)$ gauge groups. The number of $U(1)$'s per model are plotted in Figure 5.8. It is clear that most models have multiple $U(1)$ factors, and that there are more $U(1)$ factors per model for this data set than the O2L1 data set. The number of non-Abelian singlets are plotted in Figure 5.9. As with the order-2 extensions, the distribution of non-Abelian singlets drops off sharply after 0, indicating the NAHE-base single-layer models do not have a tendency to produce many non-Abelian singlets.

5.3.2 Without $\vec{S}$

Removing $\vec{S}$ from the NAHE set for the order-2 layer-1 extensions had interesting consequences on the available matter sectors. For the order-3 layer-1 extensions the effect is expected to be more drastic, as linear independence prevented any models with the same left mover as $\vec{S}$ to exist in a model. There should be less of an impact on the ST SUSY, however, since order-3 basis vectors with massless left movers always produce their own gravitino generating sector. The lower number of

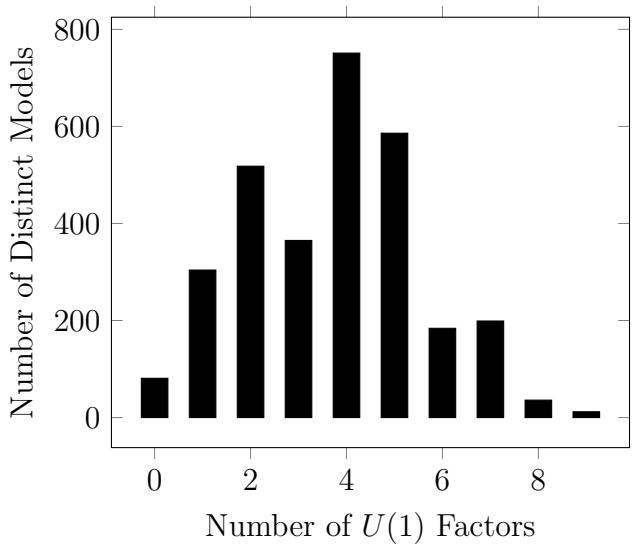


Figure 5.8: The number of $U(1)$ factors for the NAHE + O3L1 data set.

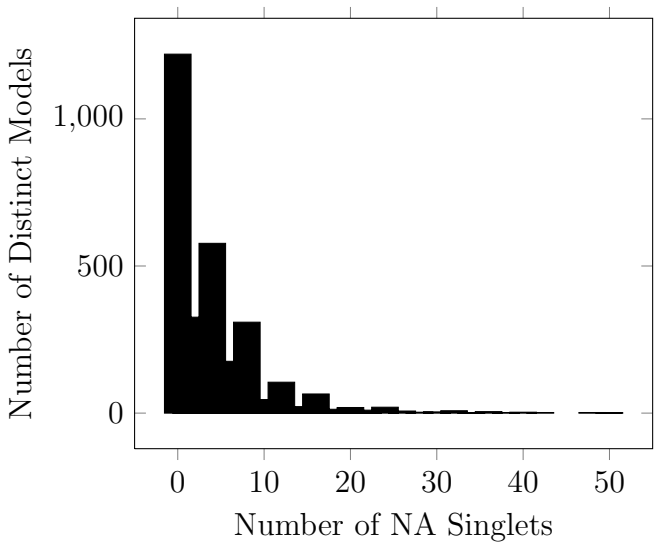


Figure 5.9: The number of non-Abelian singlets for the NAHE + O3L1 data set.

possibilities for k_{ij} values will also have an effect on the number of models in the set.

There are 447 unique models in this set out of 870,688 consistent models, and of those 146 models also belong to the data set with $\vec{S}$. Based on the number of duplicates in the O2L1 data set (without $\vec{S}$), the estimated systematic uncertainty for these statistics is 10%. There is significantly more overlap between the two sets, yet there are also significantly fewer unique models. The gauge group content of the NAHE+O3L1 without $\vec{S}$ is presented in Table 5.12. A brief comparison between Tables 5.12 and 5.10 makes it clear the presence of $\vec{S}$ did not significantly affect the gauge groups, as was the case with the order-2 extensions. For completeness the occurances of the GUT groups and other relevant statistics for this data set will be presented. The occurrances of the GUT groups in this data set are tabulated in Table 5.13. The number of gauge group factors, $U(1)$ factors, ST SUSYs, and non-Abelian singlets are plotted in Figure 5.10.

5.4 Models With GUT Groups

The next several sections will outline statistics on models containing GUT groups from the NAHE + O2L1 and NAHE + O3L1 data sets (with $\vec{S}$). The GUT groups to be examined are E_6, $SO(10)$, $SU(5)\otimes U(1)$, $SU(4)\otimes SU(2)\otimes SU(2)$ (Pati-Salam), $SU(3)\otimes SU(2)\otimes SU(2)$ (Left-Right Symmetric), and $SU(3)\otimes SU(2)\otimes U(1)$ (MSSM). In addition to the spread of statistics presented in sections 5.2 and 5.3, the number of chiral fermion generations will be counted, along with any exotics which carry observable sector charge. These statistics will be gathered for all possible observable sector configurations. Thus, if there is more than one copy of any GUT group in a model (as is often the case), all possible choices for each group forming the observable sector will be examined.

Table 5.12: The gauge group content of the NAHE + O3L1 data set without $\vec{S}$.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	368	82.33%
$SU(3)$	128	28.64%
$SU(4)$	313	70.02%
$SU(5)$	70	15.66%
$SU(6)$	96	21.48%
$SU(7)$	26	5.817%
$SU(8)$	71	15.88%
$SU(9)$	15	3.356%
$SU(10)$	14	3.132%
$SU(11)$	3	0.6711%
$SU(12)$	8	1.79%
$SU(13)$	1	0.2237%
$SU(14)$	1	0.2237%
$SO(8)$	124	27.74%
$SO(10)$	97	21.7%
$SO(12)$	53	11.86%
$SO(14)$	49	10.96%
$SO(16)$	42	9.396%
$SO(18)$	3	0.6711%
$SO(20)$	6	1.342%
$SO(22)$	1	0.2237%
$SO(24)$	2	0.4474%
$SO(26)$	1	0.2237%
E_6	37	8.277%
E_7	32	7.159%
E_8	16	3.579%
$U(1)$	430	96.2%

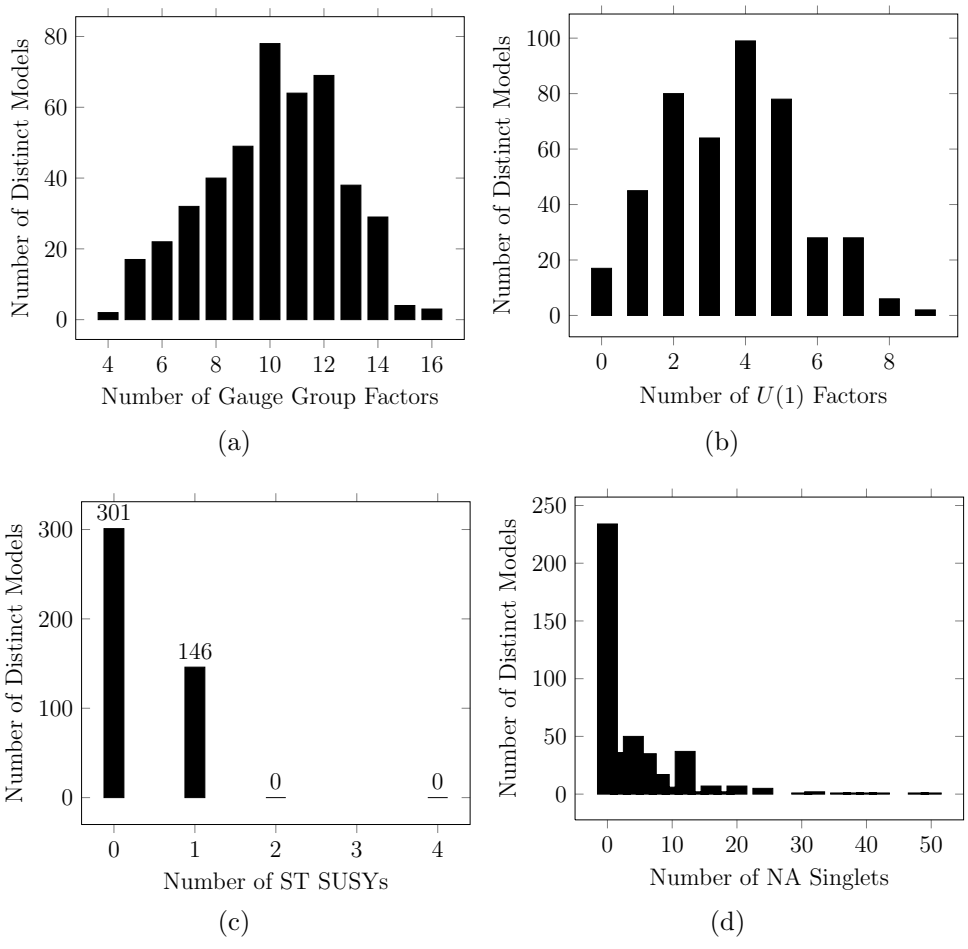


Figure 5.10: Statistics for the NAHE + O3L1 data set without $\vec{S}$.

Table 5.13: The occurrances of the GUT groups for the NAHE + O3L1 data set without $\vec{S}$.

Gauge Group	Number of Unique Models	% of Unique Models
E_6	37	8.277%
$SO(10)$	97	21.7%
$SU(5) \otimes U(1)$	70	15.66%
$SU(4) \otimes SU(2) \otimes SU(2)$	220	49.21%
$SU(3) \otimes SU(2) \otimes SU(2)$	81	18.12%
$SU(3) \otimes SU(2) \otimes U(1)$	100	22.37%

The definition of a chiral matter generation will be presented with each group for clarity, and only the "net" chiral generations will be counted. That is, if there are an equal number of barred and unbarred generations, the model will have no net chiral generations. This is done because any barred and unbarred generations paired together can be given VEVs at the GUT scale, thus removing them from the low energy phenomenology. Any "net" generations which cannot be paired must remain massless until the Higgs boson gains a VEV at the TeV scale. Such extra observable sector generations would have been observed experimentally, so removing them from the theory at the GUT scale is favorable for these models.

The statistics gathered on the chiral generations here are not enough to qualify a model or set of models as being "realistic." This study looks only to examine the basic components that sometimes lead to realistic models. Actually determining whether or not a model is realistic requires detailed analysis of the $U(1)$ charges, finding the superpotential, and finding the D- and F-flat directions. Progress is being made to automate the above steps and integrate the deeper phenomenological components of WCFFHS model building into the FF Framework. For the present

analysis, however, discussion of these aspects of WCFFHS phenomenology is omitted.

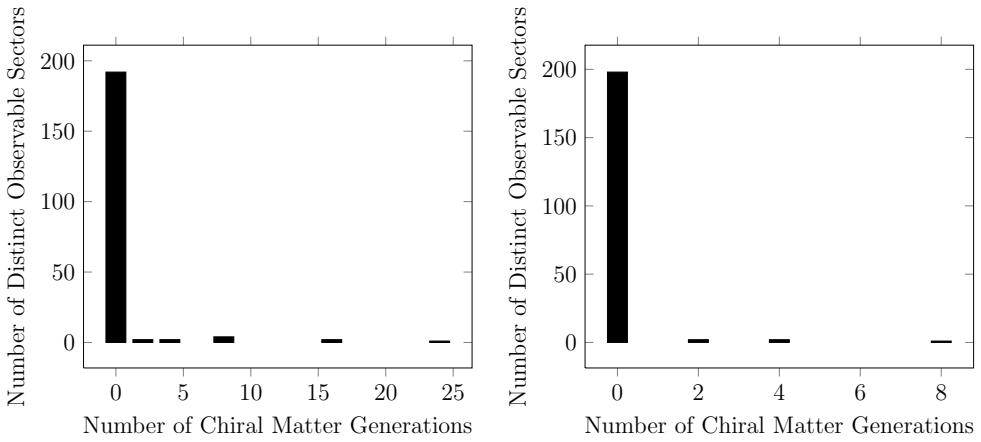
5.4.1 E_6 Models

Each SM generation of fermions fits into a 27 dimensional representation of E_6, so the number of net chiral matter generations is given by

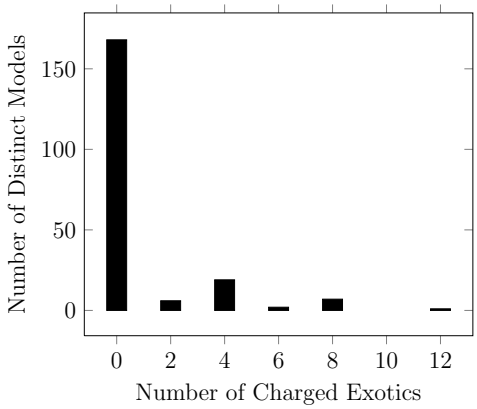
$$|N_{27} - N_{\overline{27}}|. \tag{5.1}$$

Additionally, for large GUT groups, states which transform under the hidden sector can be treated as being multiple copies of an observable generation. The hidden sector groups can be broken somewhat easily by adding basis vectors, so for certain GUT groups the dimension of the hidden sector charge is treated as the number of duplicate observable generations.

Statistics will now be presented for E_6 models coming from single layer extensions to the NAHE set. There was only one model with an E_6 group in the NAHE + O2L1 data set, but there were 193 models with E_6 in the NAHE + O3L1 data set. The number of net chiral fermion generations with and without hidden sector duplicates is plotted in Figure 5.11 along with the number of observable sector charged exotics. The hidden sector gauge group content of these models is presented in Table 5.14. Figures 5.11 shows that the distribution of net fermion generations tends to be at zero; either no 27's are produced, or every 27 is accompanied by a $\overline{27}$ for those models. For models with more than zero chiral fermion generations, the number of generations per model is even both with and without hidden sector duplicates. It is also apparent from Figure 5.11 that the E_6 charged fermions do not couple to the hidden sector in most of these models. These examinations make it clear that, though most models do not contain exotic states, there are never the correct number of chiral fermion generations to produce a realistic model.



(a) With Hidden Sector Duplicates (b) Without Hidden Sector Duplicates



(c)

Figure 5.11: Statistics related to the chiral matter generations for E_6 models in the NAHE + O3L1 data set.

Table 5.14: The hidden sector gauge group content of models containing E_6 within the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	165	85.49%
$SU(3)$	54	27.98%
$SU(4)$	113	58.55%
$SU(5)$	39	20.21%
$SU(6)$	43	22.28%
$SU(7)$	5	2.591%
$SU(8)$	10	5.181%
$SO(8)$	35	18.13%
$SO(10)$	41	21.24%
$SO(12)$	12	6.218%
$SO(16)$	19	9.845%
E_8	11	5.699%
$U(1)$	193	100%

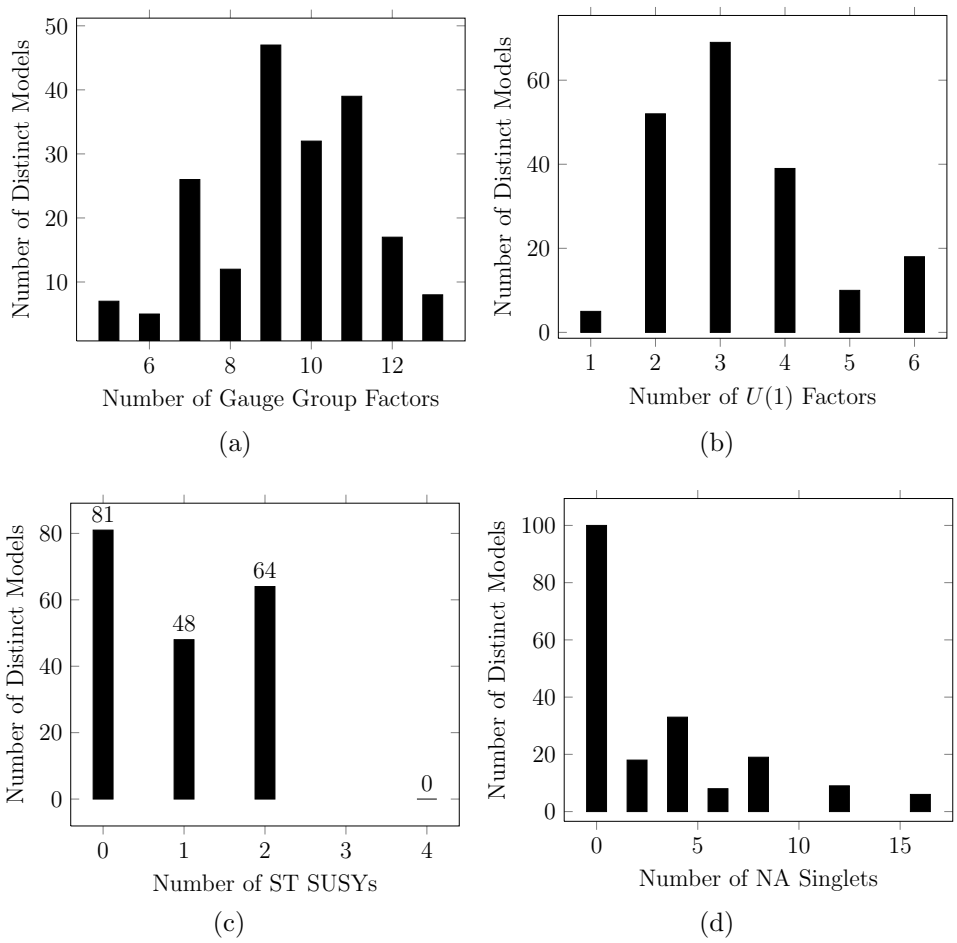


Figure 5.12: Statistics for the models containing E_6 in the NAHE + O3L1 data set.

For completeness the rest of the statistics for this data set will be presented. The number of gauge group factors per model, the number of $U(1)$ factors, the number of ST SUSYs, and the number of non-Abelian singlets are presented in Figure 5.12.

5.4.2 $SO(10)$ Models

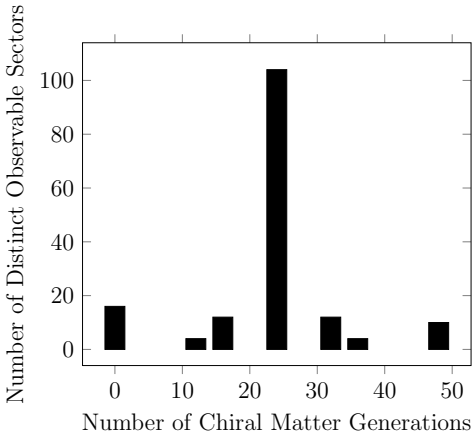
In the models containing $SO(10)$, the chiral fermion generations are defined to be in the 16 dimensional representations. Thus, the number of net chiral fermion generations is given by

$$|N_{16} - N_{\overline{16}}|. \tag{5.2}$$

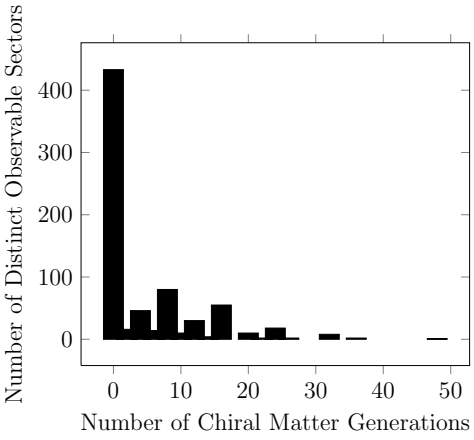
Table 5.15: Hidden sector gauge groups for $SO(10)$ models in the NAHE + O2L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	121	75.62%
$SU(2)^{(2)}$	25	15.62%
$SU(4)$	111	69.38%
$SO(5)$	59	36.88%
$SO(8)$	1	0.625%
$SO(14)$	1	0.625%
$SO(16)$	56	35%
$SO(20)$	1	0.625%
$SO(22)$	1	0.625%
E_7	51	31.87%
E_8	51	31.87%
$U(1)$	102	63.75%

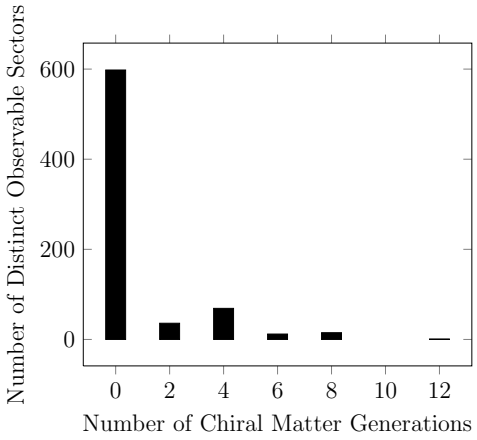
The NAHE + O2L1 data set has 160 models with $SO(10)$, while the NAHE + O3L1 data set has 659 models. The number of chiral fermion generations with hidden sector charges is plotted in Figure 5.13 for the NAHE + O2L1 and NAHE + O3L1 data sets. The number of chiral fermion generations without hidden sector charges is also plotted in Figure 5.13 for the order-3 $SO(10)$ models. There were no order-2 $SO(10)$ models with more than zero chiral fermion generations. The hidden sector gauge groups for the NAHE + O2L1 $SO(10)$ models are presented in Table 5.15. The NAHE + O3L1 $SO(10)$ model hidden sector gauge groups are presented in Table 5.16. The number of observable exotic states for these models is plotted in Figure 5.14 for the order-2 and order-3 models. As with the E_6 data sets, there



(a) NAHE + O2L1 with Hidden Sector Duplicates



(b) NAHE + O3L1 with Hidden Sector Duplicates

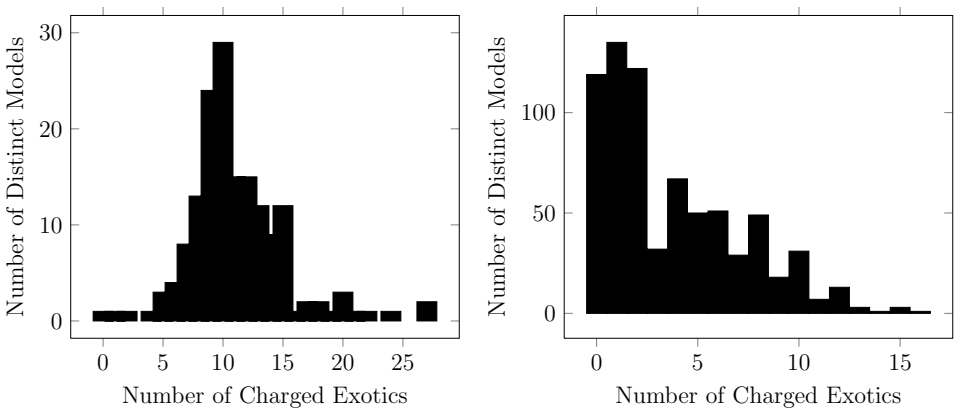


(c) NAHE + O3L1 without Hidden Sector Duplicates

Figure 5.13: Statistics for the chiral matter generations of the $SO(10)$ models in the NAHE + O2L1 and NAHE + O3L1 data sets.

Table 5.16: Hidden sector gauge groups for $SO(10)$ models in the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	570	86.49%
$SU(3)$	126	19.12%
$SU(4)$	456	69.2%
$SU(5)$	74	11.23%
$SU(6)$	132	20.03%
$SU(7)$	20	3.035%
$SU(8)$	64	9.712%
$SU(9)$	5	0.7587%
$SU(10)$	15	2.276%
$SU(12)$	12	1.821%
$SO(8)$	105	15.93%
$SO(12)$	75	11.38%
$SO(14)$	34	5.159%
$SO(16)$	69	10.47%
$SO(20)$	13	1.973%
$SO(22)$	5	0.7587%
E_6	41	6.222%
E_7	29	4.401%
E_8	20	3.035%
$U(1)$	604	91.65%



(a) NAHE + O2L1 (b) NAHE + O3L1

Figure 5.14: The number of observable sector charged exotics for $SO(10)$ models in the NAHE + O2L1 and NAHE + O3L1 data sets.

are no models with three chiral generations. The number of generations in any of the models presented is either zero or even. The distribution of charged exotics is more spread out than it was for the E_6 models. This is an artifact of the NAHE set gauge group; the $SO(10)$ states in that set are charged under the three $SU(4)$ gauge groups. Most observable $SO(10)$ states tend to keep some of those charges under the hidden sector. The remaining statistics for these models are presented in Figure 5.15 for the order-2 models. The remaining statistics for the order-3 models with $SO(10)$ are presented in Figure 5.16.

5.4.3 $SU(5) \otimes U(1)$ Models

The (flipped) $SU(5) \otimes U(1)$ GUT group's matter generations are split into multiple representations of the $SU(5)$ group. An anti-lepton doublet and the up-type quarks are placed in a $\overline{5}$ representation, while the right-handed neutrino, the anti-quark doublet, and the down-type quarks appear in a 10 dimensional representation of $SU(5)$. Thus, a generation is formed by pairing the 10-reps with the $\overline{5}$-reps. The net number of generations for an $SU(5) \otimes U(1)$ model is given by

$$|\min(N_{10}, N_{\overline{5}}) - \min(N_{\overline{10}}, N_5)|. \tag{5.3}$$

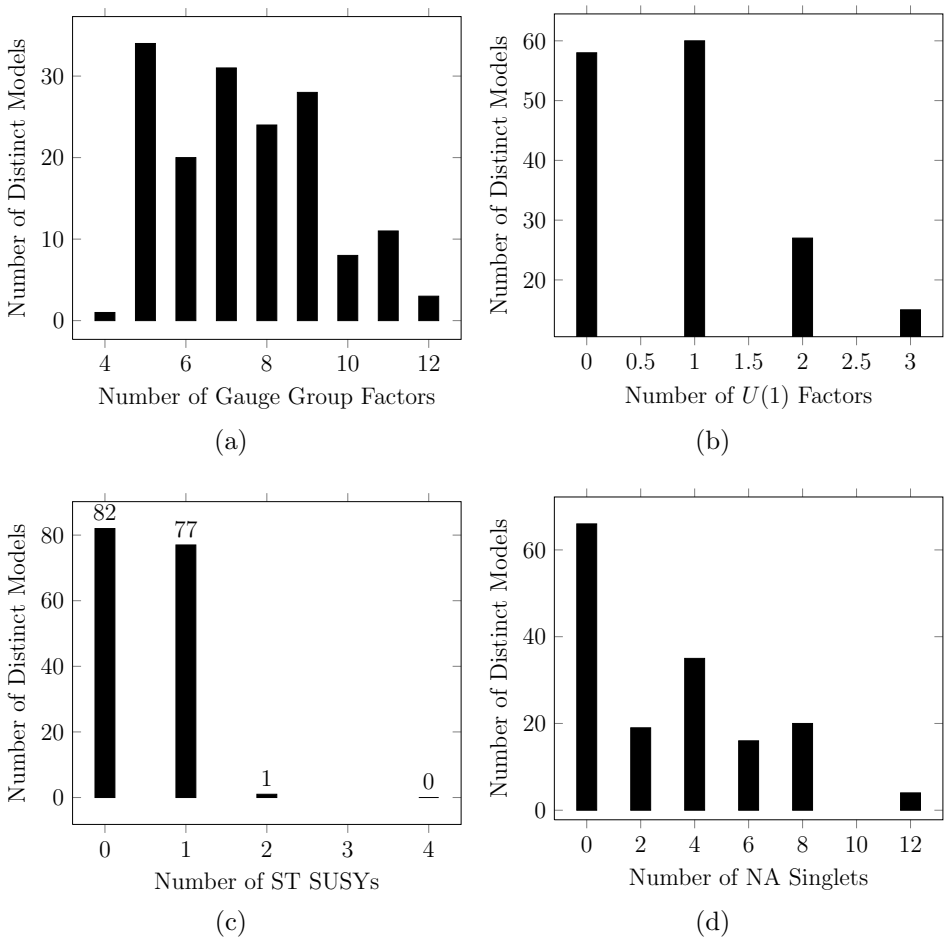


Figure 5.15: Statistics for the $SO(10)$ models in the NAHE + O2L1 data set.

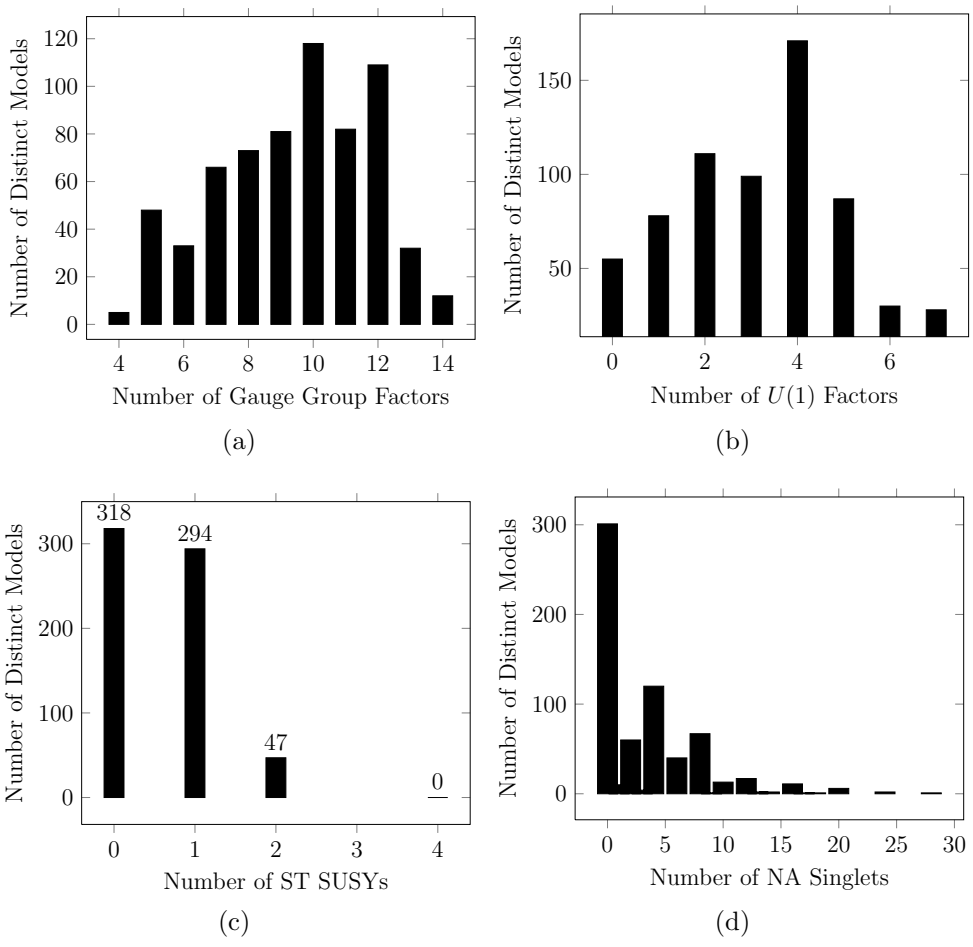
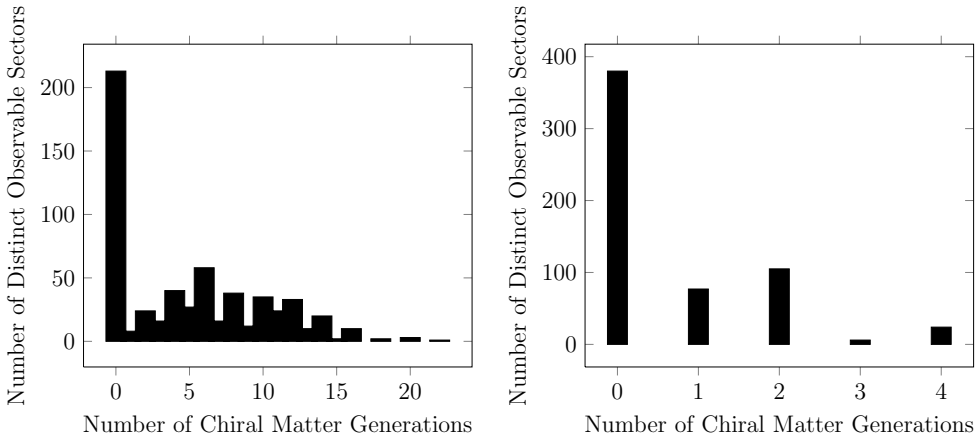
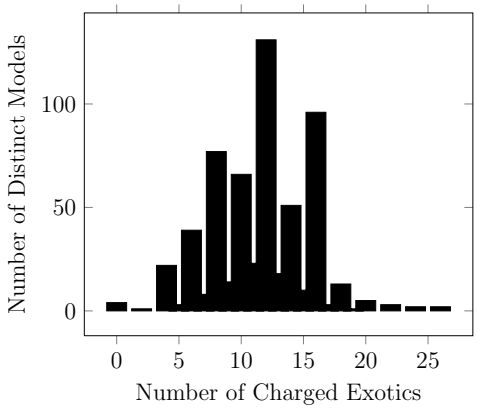


Figure 5.16: Statistics for the $SO(10)$ models in the NAHE + O3L1 data set.



(a) With Hidden Sector Duplicates (b) Without Hidden Sector Duplicates



(c)

Figure 5.17: Statistics related to observable matter for the $SU(5) \otimes U(1)$ models in the NAHE + O3L1 data set.

There were no order-2 NAHE based models containing $SU(5) \otimes U(1)$, but there were 543 order-3 models with this GUT group. The hidden sector gauge groups of those models are presented in Table 5.17. The number of net chiral generations with and without hidden sector duplicates are plotted in Figure 5.17, along with the number of exotic states with observable sector charges. The most striking feature of this data set is that there are models with three chiral generations both with and without hidden sector duplicates. The significance of this finding is that these models do not have rank cuts, and thus carry a geometric interpretation. This implies that

Table 5.17: The hidden sector gauge groups of the $SU(5) \otimes U(1)$ models in the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	449	82.69%
$SU(3)$	468	86.19%
$SU(4)$	284	52.3%
$SU(6)$	52	9.576%
$SU(7)$	50	9.208%
$SU(8)$	52	9.576%
$SU(9)$	22	4.052%
$SU(10)$	5	0.9208%
$SO(8)$	57	10.5%
$SO(10)$	74	13.63%
$SO(12)$	35	6.446%
$SO(14)$	72	13.26%
E_6	39	7.182%
E_7	10	1.842%

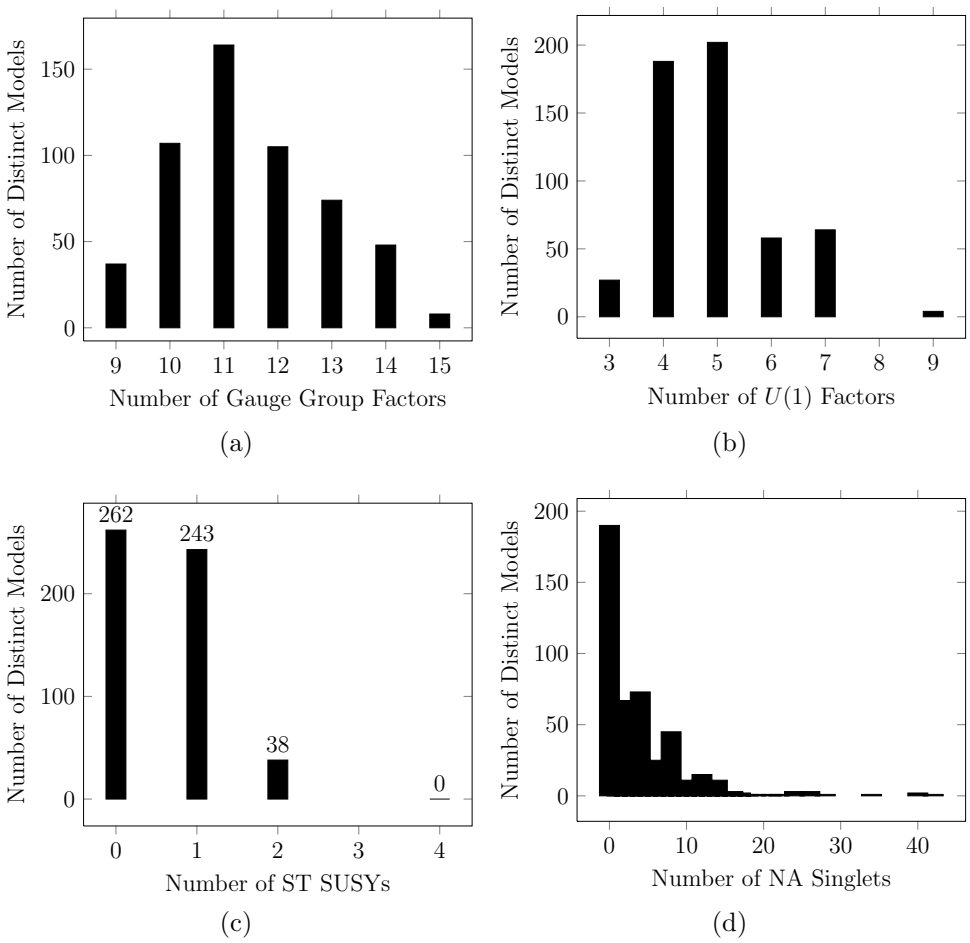


Figure 5.18: Statistics for the $SU(5) \otimes U(1)$ models in the NAHE + O3L1 data set.

these model may be written in another construction method, particularly that of orbifolding. These are the first three-generation models of their kind. More analysis will be done in Sec. 5.5 regarding this new class of three-generation models. Other statistics for this data set are plotted in Figure 5.18.

5.4.4 Pati-Salam Models

Pati-Salam models consist of models with a gauge group $SO(6) \otimes SO(4)$, which is isomorphic to the gauge group $SU(4) \otimes SU(2) \otimes SU(2)$. The latter is the form of this gauge group which appears in WCFFHS models. The quark and lepton generations are in representations of (4,2,1), while the anti-quarks and anti-leptons

are in representations of ($\overline{4}$,1,2). Because these data sets are formed by examining all permutations of possible observable sectors, the same statistics will emerge when the chiral generations and anti-generations are examined separately. As there are no three-generation models in this data set, this case will not be considered. The equation for the number of net chiral generations is

$$|N_{(4,2,1)} - N_{(\overline{4},2,1)}|. \tag{5.4}$$

There are 243 unique models containing this gauge group in the order-2 NAHE extensions, and there are 1,648 unique models with this gauge group in the order-3 NAHE extensions. The abundance of these models is expected; as they contain the most common non-Abelian gauge group, $SU(2)$, along with a group that the NAHE set model already has.

No order-2 models with the Pati-Salam gauge group have net chiral matter generations, but other statistics related to those models will be presented. Order-3 model statistics will also be presented.

For the order-2 models, the hidden sector gauge group content is presented in Table 5.18. In this data set, both $SU(2)$ at KM-level 1 and $SU(2)^{(2)}$ at KM-level 2 are included as possibilities for the observable Pati-Salam gauge group. The number of charged exotics is presented in Figure 5.19. The hidden sector gauge group content for the order-3 models is presented in Table 5.19. The number of chiral generations and observable sector charged exotics for these models are plotted in Figure 5.20.

Note that the number of chiral generations is zero in most cases for the NAHE + O3L1 data set as well. This implies that the symmetry breaking to the two $SU(2)$ gauge groups splits the three distinct generations of the NAHE observable sector evenly. The conclusion of this report is that three generation Pati-Salam models may require more complicated basis vector sets. The remaning statistics are

Table 5.18: The hidden sector gauge group content in Pati-Salam models from the NAHE + O2L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SO(5)$	60	24.69%
$SO(8)$	51	20.99%
$SO(10)$	61	25.1%
$SO(12)$	1	0.4115%
$SO(16)$	80	32.92%
$SO(20)$	2	0.823%
E_7	86	35.39%
E_8	75	30.86%
$U(1)$	185	76.13%

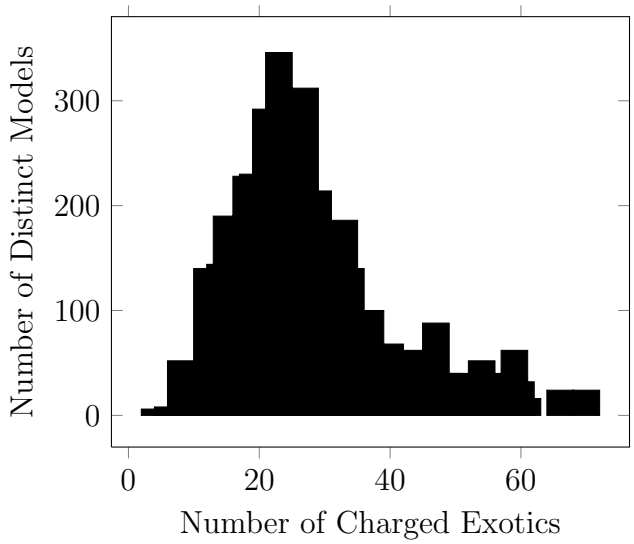


Figure 5.19: The number of observable sector charged exotics from Pati-Salam models in the NAHE + O2L1 data set.

Table 5.19: The hidden sector gauge group content of the Pati-Salam models in the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(3)$	344	20.87%
$SU(5)$	174	10.56%
$SU(6)$	414	25.12%
$SU(7)$	116	7.039%
$SU(8)$	214	12.99%
$SU(9)$	30	1.82%
$SU(10)$	31	1.881%
$SU(11)$	5	0.3034%
$SU(12)$	24	1.456%
$SO(8)$	422	25.61%
$SO(10)$	332	20.15%
$SO(12)$	255	15.47%
$SO(14)$	163	9.891%
$SO(16)$	106	6.432%
$SO(20)$	33	2.002%
E_6	81	4.915%
E_7	56	3.398%
E_8	15	0.9102%
$U(1)$	1615	98%

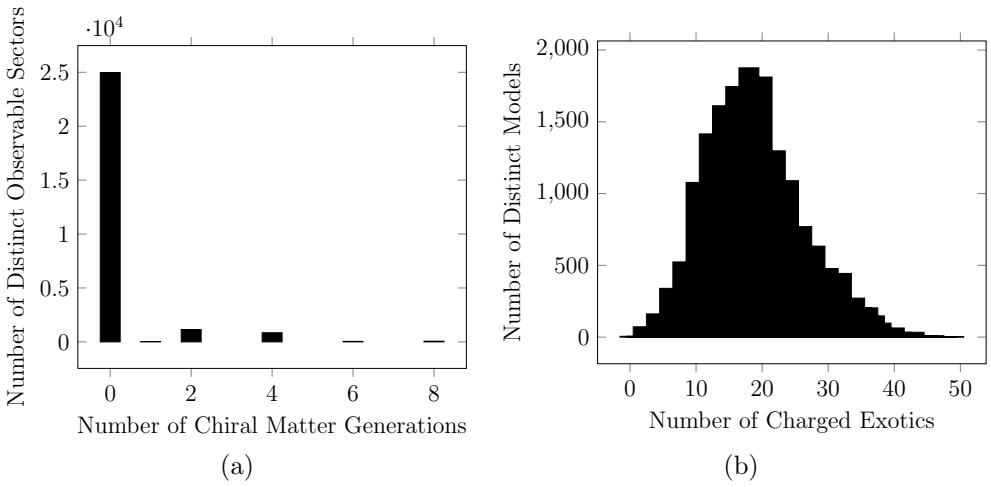


Figure 5.20: Statistics related to observable matter in the Pati-Salam models from the NAHE + O3L1 data set.

presented in Figure 5.21. Statistics for the NAHE + O3L1 data set are presented in Figure 5.22.

5.4.5 Left-Right Symmetric Models

The final GUT considered in this study is a derivative of the Pati-Salam GUT group referred to as the Left-Right Symmetric group. It retails the dual-$SU(2)$ nature of the Pati-Salam GUT, but the $SU(4)$ gauge group is broken into an $SU(3)$ group directly representing the strong force. The generations of quarks fit into a (3,2,1)-dimensional representation while the generations of anti-quarks fit into a $(\overline{3},1,2)$-dimensional representation. The lepton and anti-lepton generations are placed in a (1,2,1) and (1,1,2) representation, respectively. As the quark generations are usually more constraining in WCFFHS models, the term chiral matter generation refers only to the quarks, while the term chiral anti-generation refers only to the anti-quarks. Lepton generations will need to be taken into account when considering a quasi-realistic model, but here statistics will be gathered only with respect to the quark generations for simplicity.

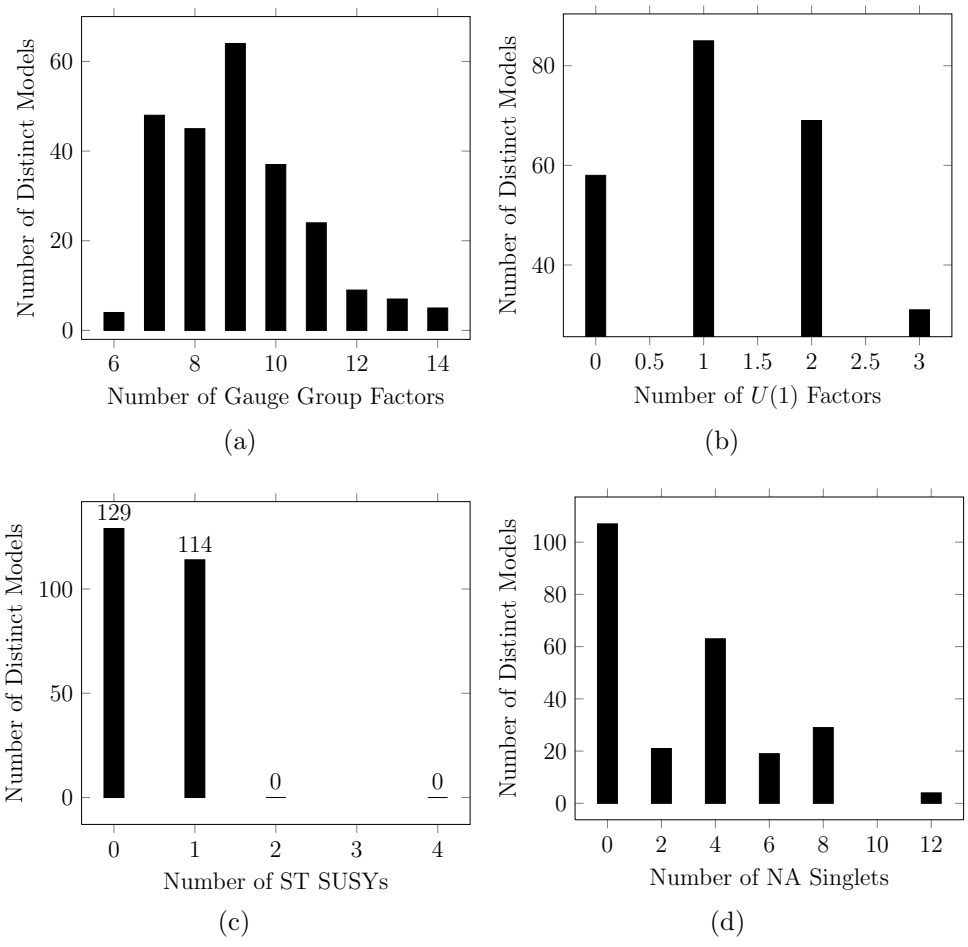


Figure 5.21: Statistics for the Pati-Salam models in the NAHE + O2L1 data set.

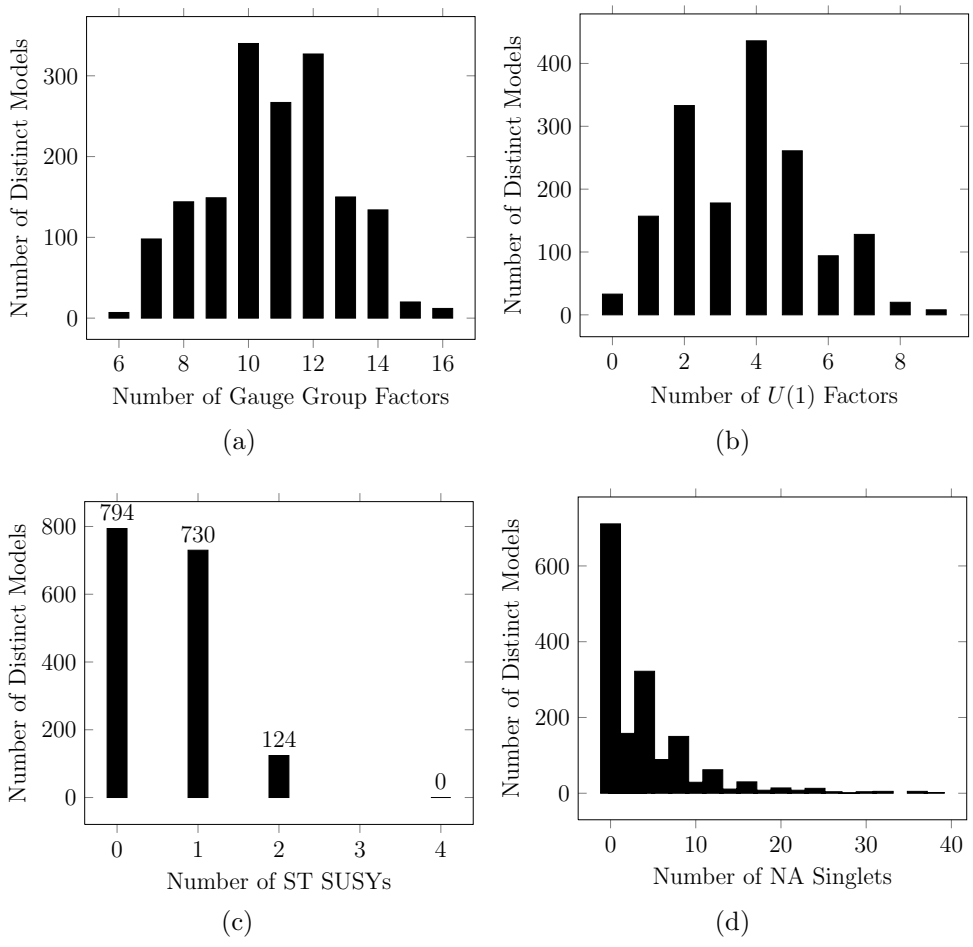


Figure 5.22: Statistics for the Pati-Salam models in the NAHE + O3L1 data set.

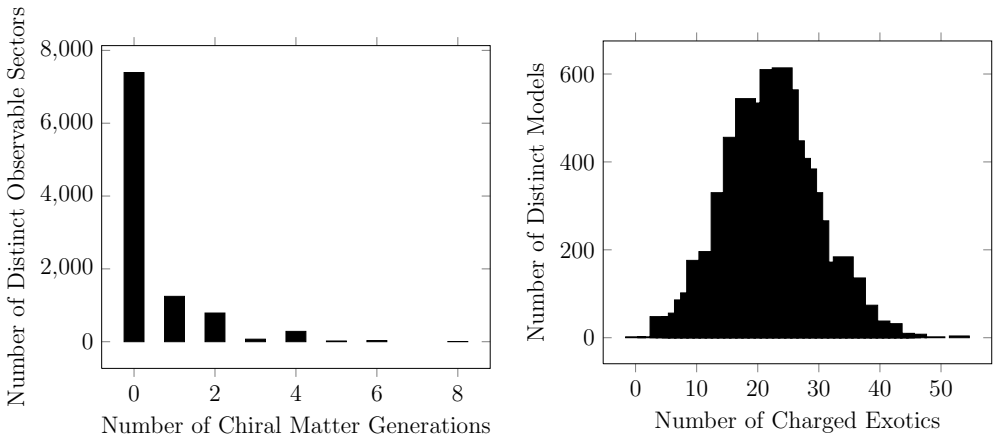


Figure 5.23: Observable matter statistics for the Left-Right Symmetric models in the NAHE + O3L1 data set.

The number of net chiral (anti)generations is given by

$$|N_{(3,2,1)} - N_{(\bar{3},2,1)}|. \tag{5.5}$$

Since the statistics loop over all possible observable sector configurations, the statistical data on the net number of chiral generations and anti-generations are identical. There are no models with the gauge group in the NAHE + O2L1 data set, as that data set contains no models with $SU(3)$ gauge groups. There are 628 distinct models in the NAHE + O3L1 data set with this gauge group. The hidden sector gauge content of those models is presented in Table 5.20. The number of net chiral generations is presented in Figure 5.23 along with the number of observable sector charged exotics. Like the $SU(5) \otimes U(1)$ data set, there are three-generation models present here. There are 70 models with three net chiral generations. One such model will be presented as an example at the end of the chapter. The remaining statistical information on these models is presented in Figure 5.24.

Table 5.20: The hidden sector gauge group content of the Left-Right Symmetric models in the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(4)$	344	54.78%
$SU(5)$	264	42.04%
$SU(6)$	112	17.83%
$SU(7)$	149	23.73%
$SU(8)$	84	13.38%
$SU(9)$	29	4.618%
$SU(10)$	10	1.592%
$SU(11)$	17	2.707%
$SO(8)$	89	14.17%
$SO(10)$	95	15.13%
$SO(14)$	53	8.439%
E_6	41	6.529%
$U(1)$	628	100%

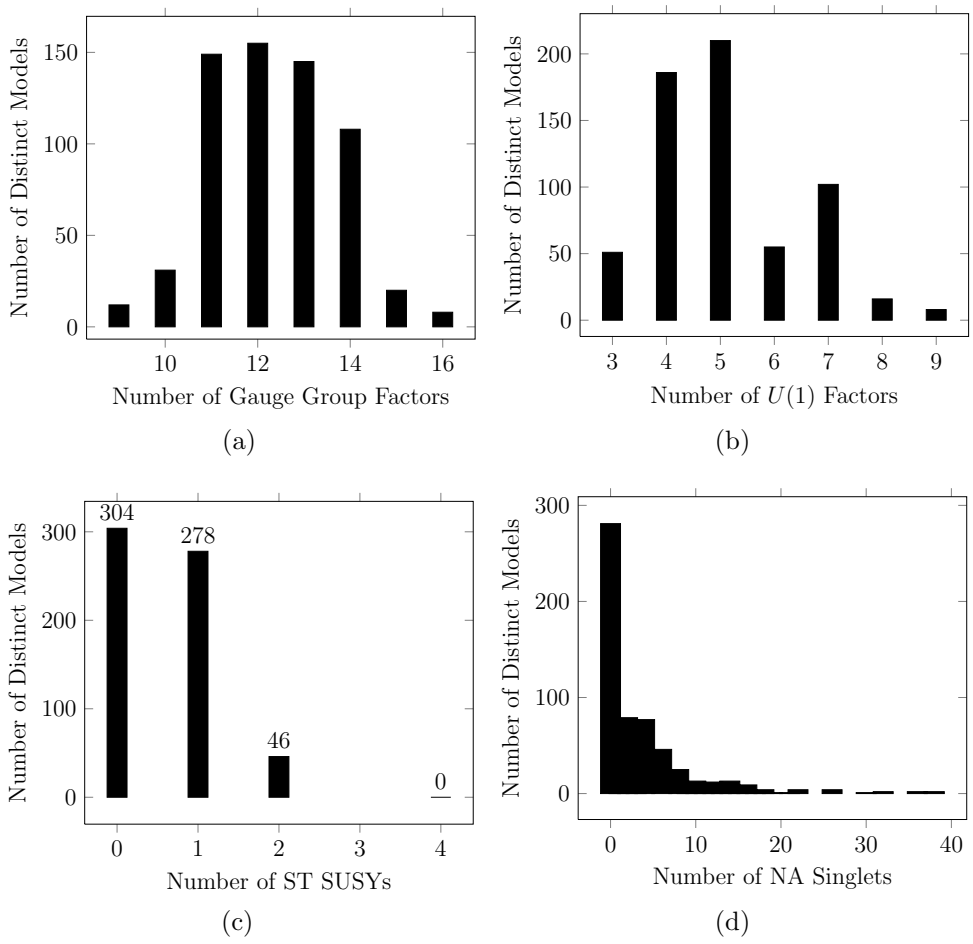


Figure 5.24: Statistics for the Left-Right Symmetric models in the NAHE + O3L1 data set.

5.4.6 MSSM-like Models

The MSSM[1] gauge group is $SU(3) \otimes SU(2) \otimes U(1)$. A generation of quarks fit in a (3,2) representation of these groups. The leptons fit in a (1,2) representation. The generations of antimatter are charged differently, as the antiparticles do not have isospin. A generation of antimatter consists of two ($\overline{3}$,1) representations, one for the "up"-type quarks and one for the "down"-type quarks. While the leptons fit into a (1,2) representation, the anti-leptons are (1,1) singlets. As was the case with Left-Right Symmetric models, the terms chiral generation and anti-generation refer only to the quarks. While the lepton generations must also be considered, statistics are only gathered for the quark generations, as they are more constraining.

The equation for the number of net chiral matter generations is

$$|N_{(3,2)} - N_{(\overline{3},2)}|, \tag{5.6}$$

while the number of net chiral antimatter generations is

$$|N_{(3,1)} - N_{(\overline{3},1)}|. \tag{5.7}$$

There are no models with this gauge group from the NAHE + O2L1 data set since there are no $SU(3)$ gauge groups. There are, however, 775 models in the NAHE + O3L1 data set with the MSSM group. The hidden sector gauge group content of models containing the MSSM gauge group is presented in Table 5.21. The number of net chiral generations and anti-generations are presented in Figure 5.25. The number of observable sector charged exotics is also plotted in Figure 5.25. There are models with three chiral matter generations, as well as models with three anti-generations. However, none of the models have three generations of quarks and anti-quarks. While these findings are still significant due to their novelty, they do not point towards phenomenologically realistic models as the $SU(5) \otimes U(1)$ and Left-

[1] Here MSSM refers only to the gauge group content. Models with this gauge group may or may not have ST SUSY.

Table 5.21: The hidden sector gauge group content of the MSSM models in the NAHE + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(4)$	412	53.16%
$SU(5)$	374	48.26%
$SU(6)$	112	14.45%
$SU(7)$	169	21.81%
$SU(8)$	112	14.45%
$SU(9)$	41	5.29%
$SU(10)$	10	1.29%
$SU(11)$	17	2.194%
$SO(8)$	97	12.52%
$SO(10)$	111	14.32%
$SO(12)$	35	4.516%
$SO(14)$	68	8.774%
E_6	46	5.935%

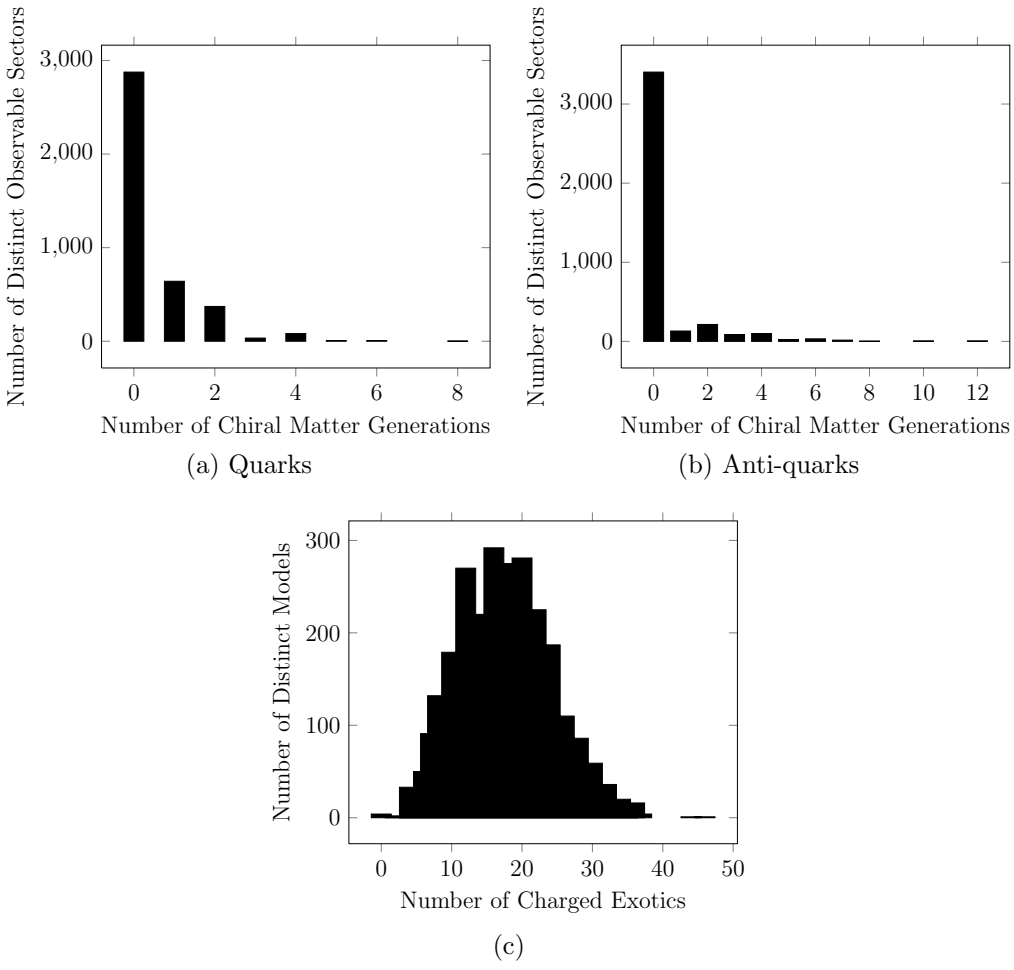


Figure 5.25: Observable matter related statistics for the MSSM models in the NAHE + O3L1 data set.

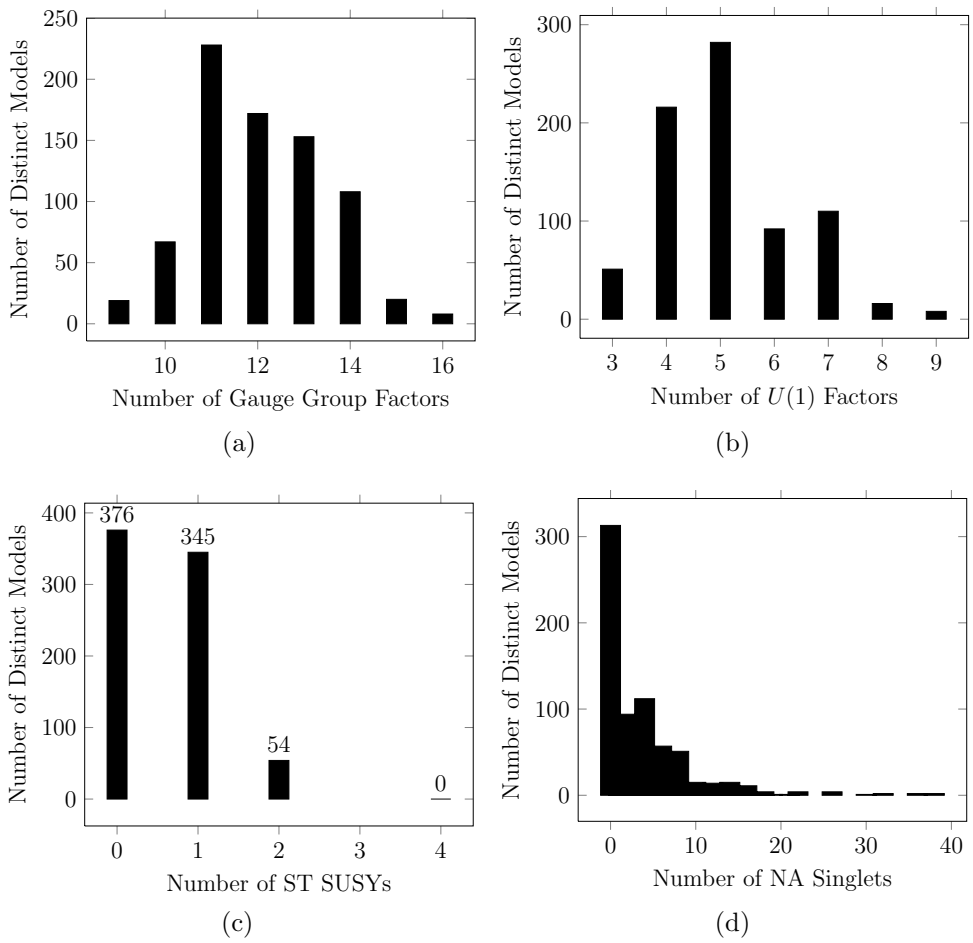
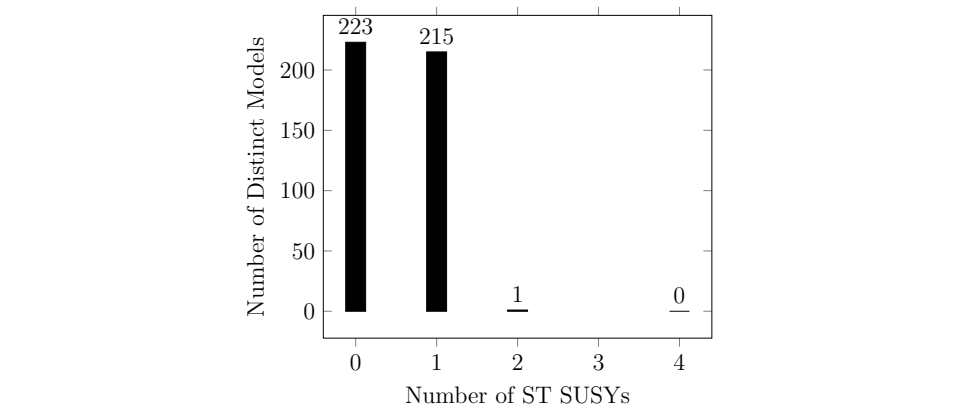


Figure 5.26: Statistics for MSSM models in the NAHE + O3L1 data set.

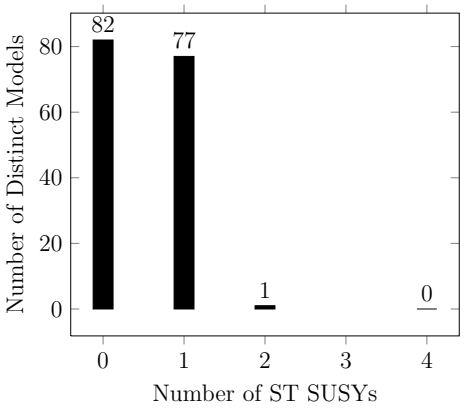
Right Symmetric models do. The remaining statistics for these models are presented in Figure 5.26.

5.4.7 ST SUSYs

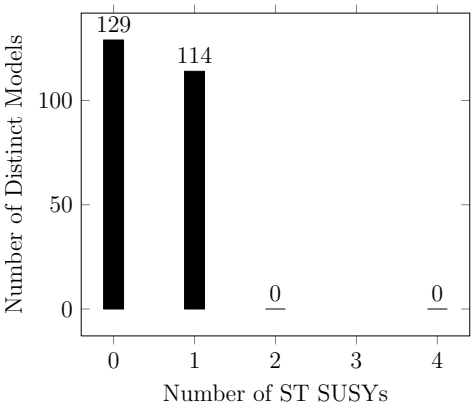
There is a trend regarding the number of ST SUSYs in GUT models — that distributions of ST SUSYs for the most part do not change. Figure 5.27 contains the ST SUSY distributions for the full data set, the $SO(10)$ models, and the Pati-Salam models. It is clear that the ST SUSY distributions are relatively even for each of the sample sets of models. The same can be said of order-3 models, whose ST SUSY distributions are presented in Figures 5.28 and 5.29. Only the E_6 models display any



(a) Full data set

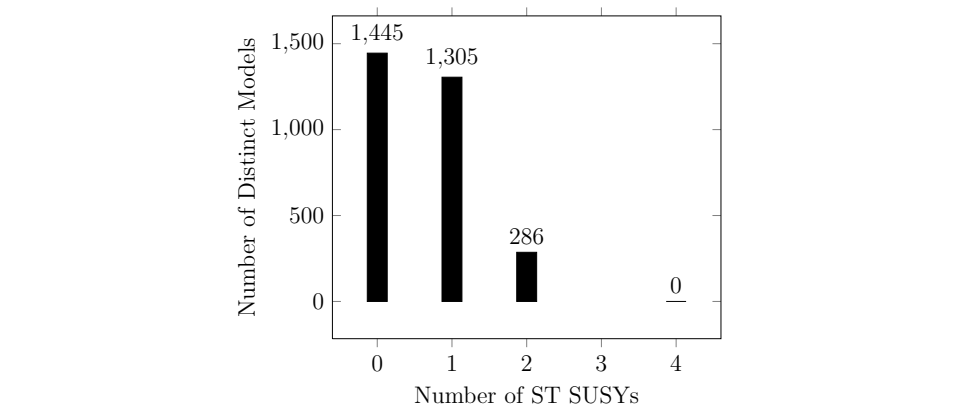


(b) $SO(10)$ models

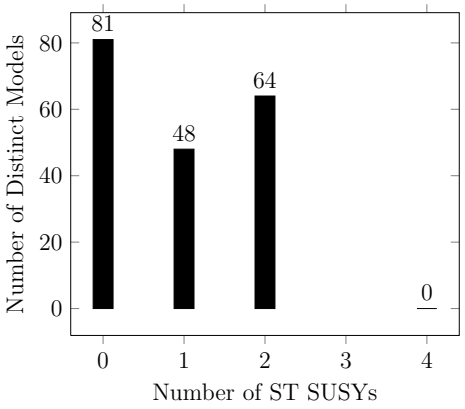


(c) Pati-Salam models

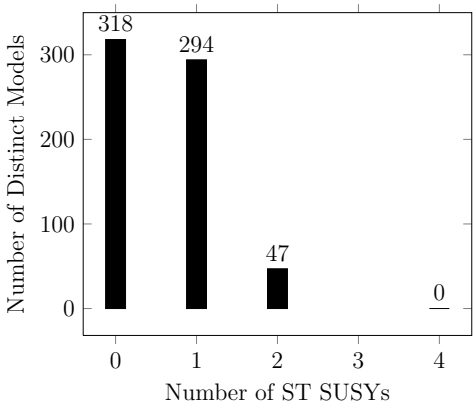
Figure 5.27: The distributions of ST SUSYs for the NAHE + O2L1 GUT group data sets.



(a) Full data set



(b) E_6 models



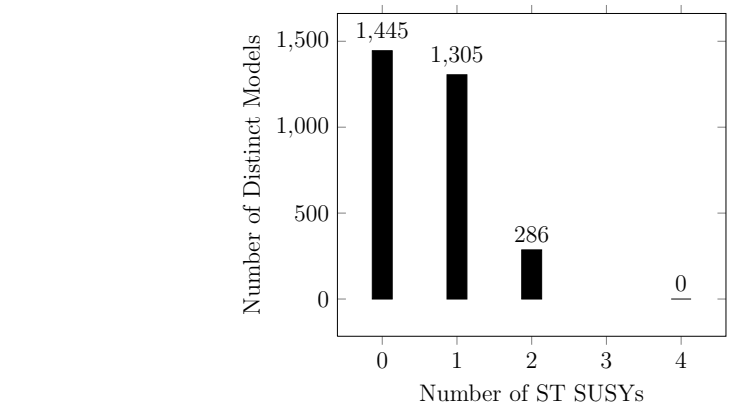
(c) $SO(10)$ models

Figure 5.28: Some of the distributions of ST SUSYs for the NAHE + O3L1 GUT group data sets.

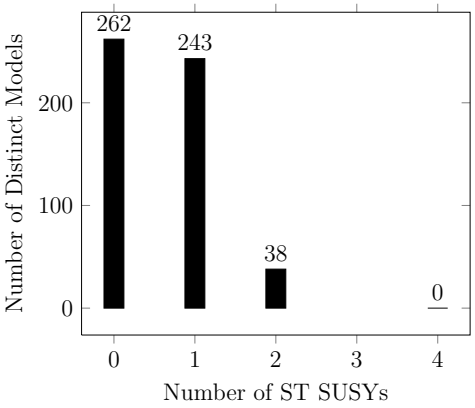
sort of statistical coupling to the number of ST SUSYs, having significantly more N=2 models than any other data set. The other samples, however, have nearly identical distributions, suggesting that the number of ST SUSYs is not statistically linked to the GUT group content.

5.5 Three Generation Models With a Geometric Interpretation

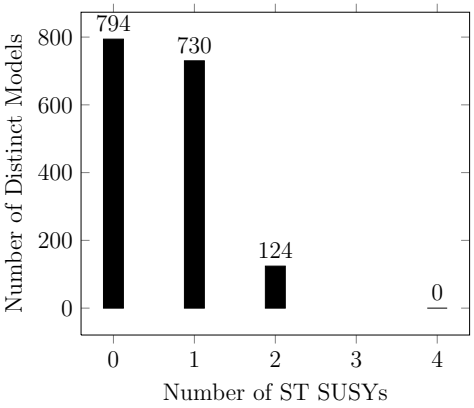
Several models containing three net chiral matter generations with $SU(5) \otimes U(1)$ and Left-Right Symmetric GUT groups were found in the NAHE + O3L1 data set. As previously mentioned, this finding is novel because these models do not have rank-cuts, and thus have a geometric interpretation. The usual statistics



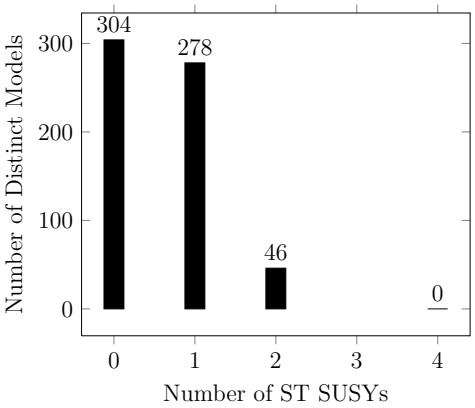
(a) Full data set



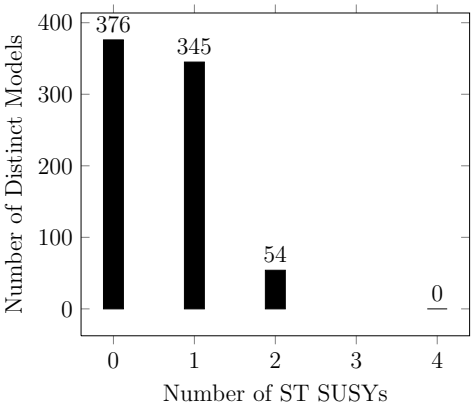
(b) $SU(5) \otimes U(1)$



(c) Pati-Salam models



(d) Left-Right Symmetric models



(e) MSSM models

Figure 5.29: The remaining distributions of ST SUSYs for the NAHE + O3L1 GUT group data sets.

will be reported for these models, and a potentially realistic model from each gauge group will be presented as an example. To determine the viability of these models, more phenomenology must be done. In particular, finding the $U(1)$ charges and the superpotential would be the first step, then the D- and F-flat directions can be found. If the flat directions can eliminate the anomalous $U(1)$ charge in addition to the observable sector charged exotic matter, then the model could be considered a quasi-realistic model. As software for automating such analysis has not yet been fully augmented, discussion of additional phenomenology must be deferred.

5.5.1 A Three Generation $SU(5) \otimes U(1)$ Model

Presented in this section is an explicit example of a NAHE based three-generation $SU(5) \otimes U(1)$ model with $N = 1$ ST SUSY. The gauge group for this model is $SU(3)^2 \otimes SU(4) \otimes SU(5) \otimes SU(6) \otimes U(1)^5$. Table 5.22 shows the basis

Table 5.22: A basis vector and k_{ij} matrix row which produces a three-generation $SU(5) \otimes U(1)$ model.

Sec	N_R	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,\ldots,5}$	$\overline{\eta}^{1}$	$\overline{\eta}^{2}$	$\overline{\eta}^{3}$	$\overline{\phi}^{1,\ldots,8}$
$\vec{v}$	3	1	1	0	0	$0,0,\frac{2}{3},\ldots,\frac{2}{3}$	$\frac{2}{3}$	0	$\frac{2}{3}$	$0,0,\frac{2}{3},\ldots,\frac{2}{3}$

Sec	O	$y^{1,2}w^{5,6}\|\|\overline{y}^{1,2}\overline{w}^{5,6}$	$y^{3,\ldots,6}\|\|\overline{y}^{3,\ldots,6}$	$w^{1,\ldots,4}\|\|\overline{w}^{1,\ldots,4}$
$\vec{v}$	3	$0,0,1,1\|\|0,0,0,0$	$0,0,0,0\|\|\frac{2}{3},\frac{2}{3},\frac{2}{3},\frac{2}{3}$	$0,0,1,1\|\|\frac{2}{3},\frac{2}{3},\frac{2}{3},\frac{2}{3}$

$$k_{\vec{v},j} = (0,\ 1,\ 1,\ 0,\ 1)$$

vectors, and Table 5.23 shows the particle content. The observable sector matter is tabulated in Table 5.24. There are no $(10,\overline{5})$ generations and three $(\overline{10},5)$ generations in this model, giving it three net chiral generations of matter[2] . However, counting the hidden sector charges as duplicates, there are 14 extra 5's and 8 extra

[2] Recall that the definition of barred and unbarred representations is arbitrary.

Table 5.23: Particle content for the three-generation $SU(5) \otimes U(1)$ model. This model also has five $U(1)$ groups and $N = 1$ ST SUSY.

QTY	$SU(3)$	$SU(3)$	$SU(4)$	$SU(5)$	$SU(7)$
2	3	1	1	1	1
3	3	1	1	1	7
3	3	3	1	1	1
1	1	$\overline{3}$	1	1	$\overline{7}$
1	1	$\overline{3}$	1	1	1
1	1	$\overline{3}$	1	1	7
2	1	$\overline{3}$	1	5	1
6	1	1	6	1	1
2	1	1	4	1	1
1	1	1	4	5	1
2	1	1	1	$\overline{5}$	1
2	1	1	1	1	$\overline{21}$
6	1	1	1	1	1
1	1	1	1	1	21
3	1	1	1	5	1
2	1	1	$\overline{4}$	1	1
1	1	1	$\overline{4}$	5	1
2	1	3	1	$\overline{5}$	1
1	1	3	1	1	$\overline{7}$
2	1	3	1	1	1
1	$\overline{3}$	$\overline{3}$	1	1	1
1	$\overline{3}$	1	1	$\overline{10}$	1
2	$\overline{3}$	1	1	$\overline{5}$	1
1	$\overline{3}$	1	1	1	$\overline{7}$
2	$\overline{3}$	1	1	1	1

Table 5.24: Observable sector matter states without hidden sector charges for the three-generation $SU(5) \otimes U(1)$ model.

QTY	$SU(3)$	$SU(3)$	$SU(4)$	$SU(5)$	$SU(7)$
2	1	$\overline{3}$	1	5	1
1	1	1	4	5	1
2	1	1	1	$\overline{5}$	1
3	1	1	1	5	1
1	1	1	$\overline{4}$	5	1
2	1	3	1	$\overline{5}$	1
1	$\overline{3}$	1	1	$\overline{10}$	1
2	$\overline{3}$	1	1	$\overline{5}$	1

$\overline{5}$'s. Because of the numerous $U(1)$ charges, this model is ideal for future $U(1)$ and flat direction analysis.

5.5.2 A Three Generation Left-Right Symmetric Model

Presented in this section is an explicitly constructed three-generation Left-Right Symmetric NAHE based model. The gauge group for this model is $SU(2)^2 \otimes SU(3)^2 \otimes SU(5) \otimes SO(10) \otimes U(1)^7$, and it has N=1 ST SUSY. The basis vectors for this model are presented in Table 5.25. The particle content of this model is presented in Table 5.26, and the observable matter is presented in Table 5.27.

The left- and right-handed isospin groups are denoted $SU(2)_L$ and $SU(2)_R$, respectively. The QCD group is denoted $SU(3)_C$. This model has three net generations of quarks, but no net generations of anti-quarks. Additionally there are thirty left- and right-handed lepton doublets. Other exotics include a quark triplet with left- and right-handed isospin, eight quark and ten anti-quark triplets without isospin. Thus, this model is not a favorable candidate for a quasi-realistic three-

Table 5.25: A basis vector and k_{ij} matrix row which produces a three-generation Left-Right Symmetric model.

Sec	N_R	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,...,5}$	$\overline{\eta}^{1}$	$\overline{\eta}^{2}$	$\overline{\eta}^{3}$	$\overline{\phi}^{1,...,8}$
$\vec{v}$	3	1	0	0	0	$\frac{2}{3}$,...,$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	$\frac{2}{3}$	0,...,0,$\frac{2}{3}$, $\frac{2}{3}$, $\frac{2}{3}$

Sec	O	$y^{1,2}w^{5,6}\|\|\overline{y}^{1,2}\overline{w}^{5,6}$	$y^{3,...,6}\|\|\overline{y}^{3,...,6}$	$w^{1,...,4}\|\|\overline{w}^{1,...,4}$
$\vec{v}$	3	0,0,1,1||0,0,$\frac{2}{3}$, $\frac{2}{3}$	0,0,0,0||0,0,$\frac{2}{3}$,$\frac{2}{3}$	0,0,1,1||$\frac{2}{3}$,$\frac{2}{3}$,$\frac{2}{3}$,$\frac{2}{3}$

$$k_{\vec{v},j} = (0,\ 1,\ 0,\ 0,\ 0)$$

generation model. It does serve as a proof of concept that three generation models can be built with single-layer extensions to the NAHE set, however.

5.6 Conclusions

The statistics presented in this chapter make it clear that the NAHE set does serve its intended purpose as a basis for quasi-realistic WCFFHS models at a statistical level. Three generation models were constructed from order-3 extensions to the NAHE set. A summary of the GUT group analysis is presented in Table 5.28. Two three-generation models were discussed - a flipped-$SU(5)$ model and a Left-Right Symmetric model. While they did have the requisite number of chiral matter generations, there were several unfavorable properties in both models that prevented them from being considered quasi-realistic. They are a proof that three-generation models with geometric interpretations can be built with order-3 basis vector extensions.

The distributions of ST SUSYs across the GUT group subsets remained largely the same, save the E_6 models, which displayed a greater statistical tendency for ST SUSY enhancements. It was also shown that the presence of the $\vec{S}$ did not significantly impact the gauge content. However, the matter content of the models without $\vec{S}$ is affected, as the $\vec{S}$ sector produces states other than SUSY partners.

Table 5.26: The particle content of the three-generation Left-Right Symmetric Model. This model also has 7 $U(1)$'s and $N = 1$ ST SUSY.

QTY	$SU(2)_L$	$SU(2)_R$	$SU(3)_C$	$SU(3)$	$SU(5)$	$SO(10)$
1	2	2	$\overline{3}$	1	1	1
2	2	1	3	1	1	1
2	2	1	1	3	1	1
3	2	1	1	1	1	1
3	2	1	1	1	5	1
1	2	1	1	$\overline{3}$	1	1
1	2	1	$\overline{3}$	1	1	1
2	1	2	1	3	1	1
2	1	2	1	1	$\overline{5}$	1
3	1	2	1	1	1	1
1	1	2	1	1	5	1
1	1	2	1	$\overline{3}$	1	1
3	1	2	$\overline{3}$	1	1	1
2	1	1	3	3	1	1
4	1	1	3	1	1	1
2	1	1	3	$\overline{3}$	1	1
3	1	1	1	3	1	1
2	1	1	1	1	$\overline{10}$	1
1	1	1	1	1	$\overline{5}$	1
1	1	1	1	1	1	$\overline{16}$
3	1	1	1	1	1	10
9	1	1	1	1	1	1
3	1	1	1	1	1	16
5	1	1	1	1	5	1
1	1	1	1	$\overline{3}$	$\overline{10}$	1
1	1	1	$\overline{3}$	3	1	1
1	1	1	$\overline{3}$	1	$\overline{5}$	1

Table 5.27: The observable matter content of the three-generation Left-Right Symmetric Model.

QTY	$SU(2)_L$	$SU(2)_R$	$SU(3)_C$	$SU(3)$	$SU(5)$	$SO(10)$
1	2	2	$\overline{3}$	1	1	1
2	2	1	3	1	1	1
2	2	1	1	3	1	1
3	2	1	1	1	1	1
3	2	1	1	1	5	1
1	2	1	1	$\overline{3}$	1	1
1	2	1	$\overline{3}$	1	1	1
2	1	2	1	3	1	1
2	1	2	1	1	$\overline{5}$	1
3	1	2	1	1	1	1
1	1	2	1	1	5	1
1	1	2	1	$\overline{3}$	1	1
3	1	2	$\overline{3}$	1	1	1
2	1	1	3	3	1	1
4	1	1	3	1	1	1
2	1	1	3	$\overline{3}$	1	1
1	1	1	$\overline{3}$	3	1	1
1	1	1	$\overline{3}$	1	$\overline{5}$	1

Table 5.28: A summary of the GUT group study with regard to the number of chiral fermion generations in the NAHE set investigation.

GUT	Net Chiral Generations?	Three Generations?
O2L1 $SO(10)$	Yes	No
O2L1 Pati-Salam	No	No
O3L1 E_6	Yes	No
O3L1 $SO(10)$	Yes	No
O3L1 $SU(5) \otimes U(1)$	Yes	Yes
O3L1 Pati-Salam	Yes	No
O3L1 L-R Symmetric	Yes	Yes
O3L1 MSSM	Yes	Yes

CHAPTER SIX

Preliminary Systematic NAHE Variation Extensions

While there have been many quasi-realistic models constructed from the NAHE basis, other bases can be used to create different classes of realistic and quasi-realistic heterotic string models. In this chapter, one such basis will be discussed, called the NAHE variation [52]. Like the NAHE set, the NAHE variation is a collection of five order-2 basis vectors. However, the sets of matching boundary conditions are larger than those of the NAHE set. This allows for a new class of models with "mirrored" groups - that is, with gauge groups that occur in even factors. Some also have mirrored matter representations that do not interact with one another. This mirroring means that the hidden sector content matches the observable sector content, making the dark matter identical to the observable SM. Several scenarios with mirrored dark matter have been presented as viable phenomenological descriptions of the universe [66, 67, 68, 69]. The NAHE set does not have a tendency to produce mirrored models because the boundary conditions making up the $SU(4)^3$ gauge groups break the mirroring between the elements $\overline{\psi}, \overline{\eta}$ and $\overline{\phi}$. As will be shown, the NAHE variation keeps this mirroring, allowing for mirrored gauge groups and matter representations.

6.1 The NAHE Variation

The NAHE variation is a collection of five order-2 basis vectors that generate a model with gauge group $SO(22) \otimes E_6 \otimes U(1)^5$. The basis vectors making up this set are presented in Table 6.1. The capacity for mirroring is clear from the basis vectors: $\overline{\psi}^{1,\ldots,5}$, along with $\overline{w}^{1,\ldots,6}$ (when they are placed into complex pairs), all have the boundary conditions as $\overline{\phi}^{1,\ldots,8}$. This allows the two parts of the basis vector to mirror one another, and can allow for both mirrored gauge groups and matter representations. The particle content of the NAHE variation model is presented in

Table 6.1: The basis vectors and GSO coefficients of the NAHE variation arranged into sets of matching boundary conditions. The elements ψ, $\overline{\psi}^i$, $\overline{\eta}^i$, and $\overline{\phi}^i$ are expressed in a complex basis. x^i, y^i, w^i, $\overline{y}^i$, and $\overline{w}^i$ are expressed in a real basis.

Sec	O	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,...,5}$	$\overline{\eta}^1$	$\overline{\eta}^2$	$\overline{\eta}^3$	$\overline{\phi}^{1,...,8}$
$\vec{1\!\!1}$	2	1	1	1	1	1,...,1	1	1	1	1,...,1
$\vec{S}$	2	1	1	1	1	0,...,0	0	0	0	0,...,0
$\vec{b}_1$	2	1	1	0	0	1,...,1	1	0	0	0,...,0
$\vec{b}_2$	2	1	0	1	0	1,...,1	0	1	0	0,...,0
$\vec{b}_3$	2	1	0	0	1	1,...,1	0	0	1	0,...,0

Sec	O	$y^{12}\|\|\overline{y}^{12}$	$y^{34}\|\|\overline{y}^{34}$	$y^{56}\|\|\overline{y}^{56}$	$w^{1,...,6}\|\|\overline{w}^{1,...,6}$
$\vec{1\!\!1}$	2	1||1	1||1	1||1	1,...,1||1,...,1
$\vec{S}$	2	0||0	0||0	0||0	0,...,0||0,...,0
$\vec{b}_1$	2	0||0	1||1	1||1	0,...,0||0,...,0
$\vec{b}_2$	2	1||1	0||0	1||1	0,...,0||0,...,0
$\vec{b}_3$	2	1||1	1||1	0||0	0,...,0||0,...,0

$$k_{ij} = \left(\begin{array}{c|ccccc} & \vec{1\!\!1} & \vec{S} & \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \\ \hline \vec{S} & 0 & 0 & 0 & 0 & 0 \\ \vec{b}_1 & 1 & 1 & 1 & 1 & 1 \\ \vec{b}_2 & 1 & 1 & 1 & 1 & 1 \\ \vec{b}_3 & 1 & 1 & 1 & 1 & 1 \end{array}\right)$$

Table 6.2: The particle content for the NAHE variation model. The model also has five $U(1)$ groups and $N = 1$ ST SUSY.

QTY	$SO(22)$	E_6
30	22	1
15	1	27
90	1	1
15	1	$\overline{27}$

Table 6.2. The observable sector is generally regarded as being the E_6, as it is a GUT group. However, additional observable sectors may come out of the $SO(22)$ as well. The large number of $U(1)$ groups and non-Abelian singlets (when compared to the NAHE set) is less phenomenologically favorable. However, the quantities of both can reduce drastically, as will be seen shortly when the statistics for single layer extensions are presented.

6.2 Order 2, Layer 1 Extensions

There were 309 unique models out of 1,315,328 total consistent models built given the input parameters. Approximately 2% of the models in the data set without rank cuts were duplicates, while none of the models with rank cuts had duplicates. The gauge group content of those models is presented in Table 6.3. The most common gauge group in this data set is $U(1)$, while the most common non-Abelian gauge group is $SU(2)$. However, less than half of the models in the data set contain $SU(2)$. The other pertinent feature of these gauge groups is the presence of non-simply laced gauge groups with high rank. The $SO(2N + 1)$ groups range from rank 2 up to rank 10. This is a feature unique to the NAHE variation extensions (at least within this study). The NAHE set extensions do not have B-class gauge groups with rank higher than 2. Finally, worth pointing out is how many models retain their E_6 symmetry — about one third. These models will be revisited later

Table 6.3: The gauge group content of the NAHE variation + O2L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	131	42.39%
$SU(2)^{(2)}$	18	5.825%
$SU(4)$	33	10.68%
$SU(6)$	99	32.04%
$SU(8)$	1	0.3236%
$SU(10)$	1	0.3236%
$SO(5)$	18	5.825%
$SO(7)$	12	3.883%
$SO(9)$	18	5.825%
$SO(11)$	14	4.531%
$SO(13)$	18	5.825%
$SO(15)$	12	3.883%
$SO(17)$	18	5.825%
$SO(19)$	18	5.825%
$SO(21)$	18	5.825%
$SO(8)$	30	9.709%
$SO(10)$	125	40.45%
$SO(12)$	38	12.3%
$SO(14)$	33	10.68%
$SO(16)$	33	10.68%
$SO(18)$	38	12.3%
$SO(20)$	36	11.65%
$SO(22)$	31	10.03%
$SO(24)$	2	0.6472%
$SO(32)$	1	0.3236%
E_6	101	32.69%
E_7	3	0.9709%
E_8	1	0.3236%
$U(1)$	304	98.38%

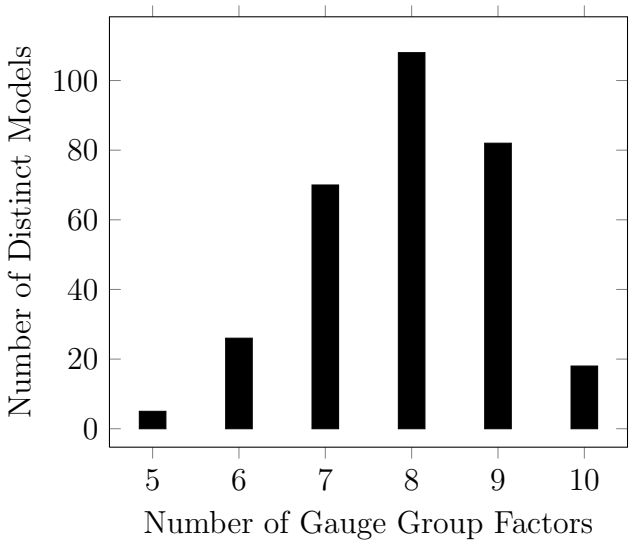


Figure 6.1: The number of gauge group factors in the NAHE variation + O2L1 data set.

with the E_6 treated as an observable sector gauge group, and the number of chiral matter generations they have will be statistically examined.

Also of interest regarding the gauge group content of this data set is the number of gauge group factors present in each model. Those are plotted in Figure 6.1. The distribution of the number of gauge group factors across the unique models in this data set have a peak around 8. This is close to the "initial" value (from the NAHE variation alone) of 7. There are a few models in which some of the factors have enhancements. These are likely the result of enhancements to the $U(1)$ groups in most cases.

Also related to the gauge group content is the distribution of the number of $U(1)$'s in this data set. That is plotted in Figure 6.2. The distribution in Figure 6.2 has a clear peak at 5, implying that many of these order-2 extensions do not alter the number of $U(1)$'s.

The frequencies of the GUT groups in this data set are tabulated in Table 6.4. Regarding the matter content, the number of ST SUSYs is plotted in Figure 6.3,

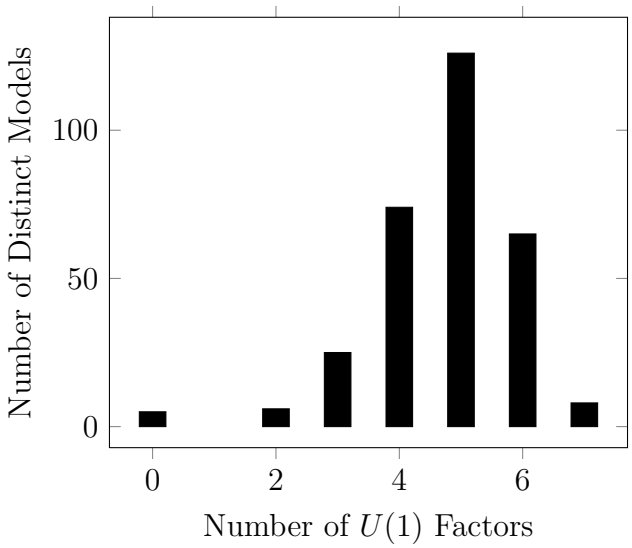


Figure 6.2: The number of $U(1)$ factors for the NAHE variation + O2L1 data set.

Table 6.4: The GUT group content of the NAHE variation + O2L1 data set.

GUT Group	Number of Unique Models	% of Unique Models
E_6	101	32.69%
$SO(10)$	125	40.45%
$SU(5) \otimes U(1)$	0	0 %
$SU(4) \otimes SU(2) \otimes SU(2)$	0	0%
$SU(3) \otimes SU(2) \otimes SU(2)$	0	0%
$SU(3) \otimes SU(2) \otimes U(1)$	0	0%

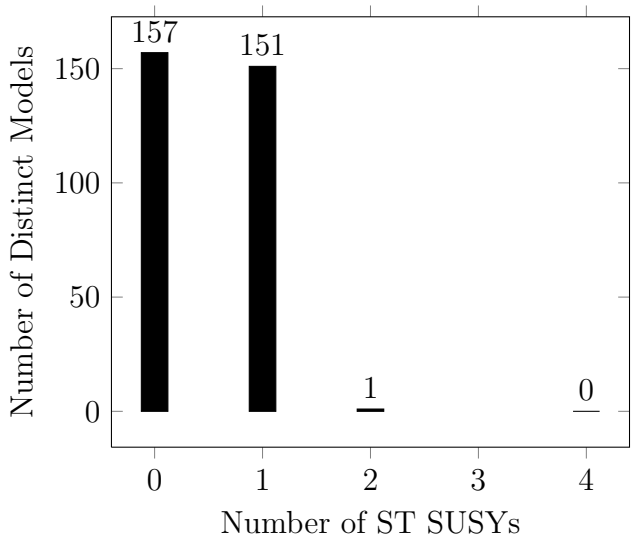


Figure 6.3: The number of ST SUSYs in the NAHE variation + O2L1 data set.

and the number of non-Abelian singlets is plotted in Figure 6.4. It is clear from Figure 6.4 that the number of non-Abelian singlets can get quite high. While most models have between 50 and 80, there can be up to 250 non-Abelian singlets in a model. This implies that many models in this data set cannot be viable candidates for quasi-realistic or realistic models.

6.3 Order 3, Layer 1 Extensions

As was the case with the NAHE set order-3 extensions, there are more distinct models in the NAHE variation + O3L1 data set. Out of 442,272 models built 1,166 of them were unique. Based on the order-2 redundancies, the systematic uncertainty for this data set is estimated to be 2%. Their gauge group content is tabulated in Table 6.5. As was the case with the O2L1 data set, $U(1)$ is the most common gauge group. However, the percentage is significantly lower here, about 86% as opposed to 98%. This suggests that some of the added basis vectors are unifying the five $U(1)$'s in the NAHE variation into larger gauge groups. Also of note is the number of models with gauge groups of higher rank than 11. In the O2L1 data set, there

Table 6.5: The gauge group content of the NAHE variation + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	731	62.69%
$SU(3)$	128	10.98%
$SU(4)$	355	30.45%
$SU(5)$	165	14.15%
$SU(6)$	167	14.32%
$SU(7)$	75	6.432%
$SU(8)$	143	12.26%
$SU(9)$	164	14.07%
$SU(10)$	169	14.49%
$SU(11)$	137	11.75%
$SU(12)$	56	4.803%
$SU(13)$	4	0.3431%
$SU(14)$	1	0.08576%
$SO(8)$	376	32.25%
$SO(10)$	271	23.24%
$SO(12)$	151	12.95%
$SO(14)$	81	6.947%
$SO(16)$	106	9.091%
$SO(18)$	28	2.401%
$SO(20)$	69	5.918%
$SO(22)$	5	0.4288%
$SO(24)$	11	0.9434%
$SO(28)$	13	1.115%
$SO(30)$	1	0.08576%
$SO(32)$	2	0.1715%
$SO(36)$	1	0.08576%
E_6	68	5.832%
E_7	24	2.058%
E_8	9	0.7719%
$U(1)$	1002	85.93%

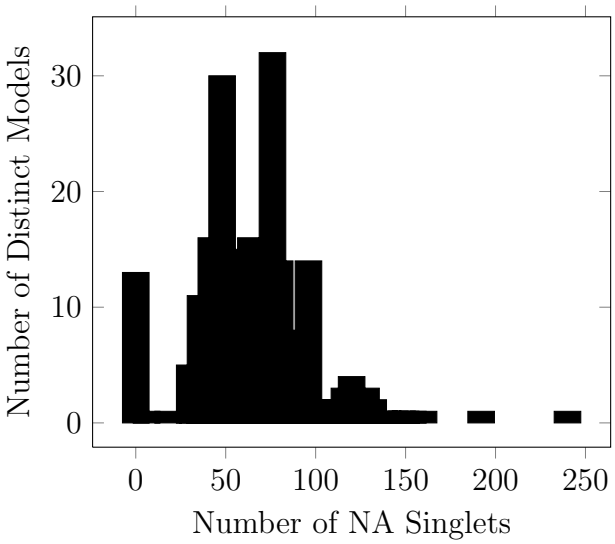


Figure 6.4: The number of non-Abelian singlets in the NAHE variation + O2L1 data set.

were only three models of this type, about 1%. In the O3L1 data set, there were 28 models with this property, about 2.4%. While it may seem from Tables 6.3 and 6.5 that the order-3 models are more prone to enhancements, Figure 6.5 makes it clear that is not the case. The distribution of the number of gauge group factors for a model peaks around 9-11 factors, as opposed to the peak around 8 factors for the order-2 models. However, there are several models with enhancements, even some models with as few as 2 distinct gauge group factors in them, something not seen with the order-2 models. This implies there is a class of order-3 basis vectors that greatly enhances the gauge group symmetries, while most order-3 models break them.

The number of $U(1)$ gauge groups per model is plotted in Figure 6.6. The distribution of $U(1)$ peaks between 5 and 7. More interestingly, a nontrivial number of models do not have $U(1)$ symmetries at all. This implies, when combined with Figure 6.5, that in some models the $U(1)$ are enhancing to larger (but still small relative to $SO(22)$ and E_6) gauge groups. The mechanism producing this effect

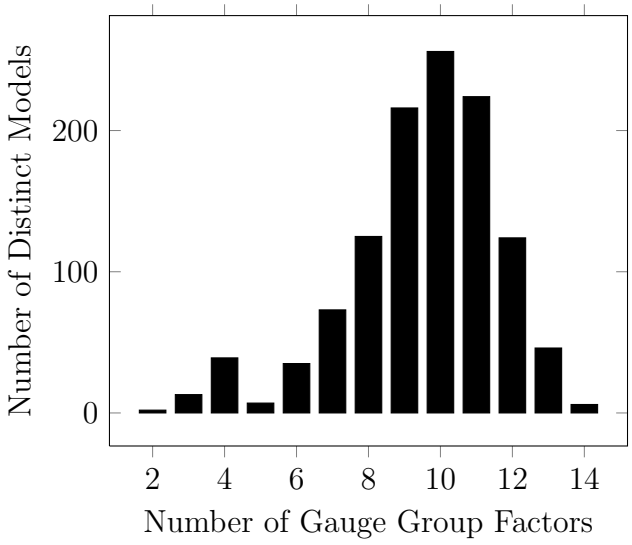


Figure 6.5: The number of gauge group factors in the NAHE variation + O3L1 data set.

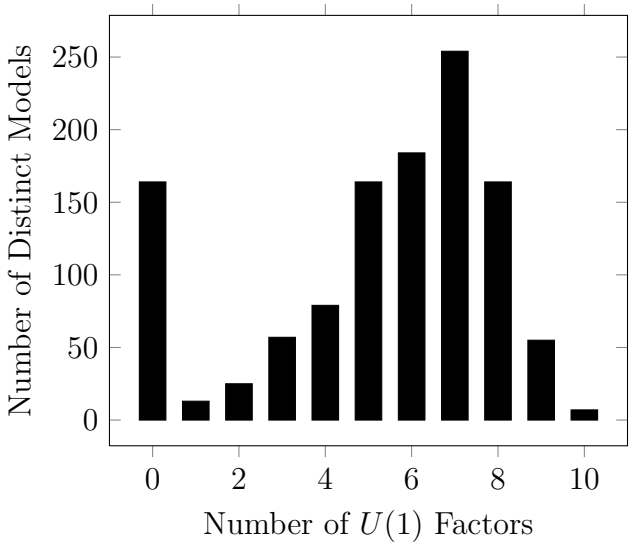


Figure 6.6: The number of $U(1)$ factors in the NAHE variation + O3L1 data set.

Table 6.6: The GUT group content of the NAHE variation + O3L1 data set.

GUT Group	Number of Unique Models	% of Unique Models
E_6	68	5.832%
$SO(10)$	271	23.24%
$SU(5) \otimes U(1)$	165	14.15%
$SU(4) \otimes SU(2) \otimes SU(2)$	125	10.72%
$SU(3) \otimes SU(2) \otimes SU(2)$	61	5.232%
$SU(3) \otimes SU(2) \otimes U(1)$	63	5.403%

warrants further study, as it could be used to reduce the number of $U(1)$ factors for order-layer combinations that tend to produce too many $U(1)$'s. The frequency of the GUT groups is presented in Table 6.6. The number of ST SUSYs is presented in Figure 6.7. While there are a statistically significant number of enhanced ST SUSYs (expected from models with odd-ordered right movers), the majority of these models have $N = 0$ ST SUSY. The number of non-Abelian singlets is plotted in Figure 6.8. The distribution of non-Abelian singlets indicates that a large number of models do not have any non-Abelian singlets. It is possible that this is related to the number of models with no $U(1)$ factors.

6.4 Models with GUT Groups

As a parallel to the NAHE set extension study, the subsets of models containing the GUT groups E_6, $SO(10)$, $SU(5) \otimes U(1)$, $SU(4) \otimes SU(2) \otimes SU(2)$ (Pati-Salam), $SU(3) \otimes SU(2) \otimes SU(2)$ (Left-Right Symmetric), and $SU(3) \otimes SU(2) \otimes U(1)$ (MSSM). Like the NAHE study, the usual statistics will be reported along with the number of net chiral generations for models containing the GUT groups in question. If there

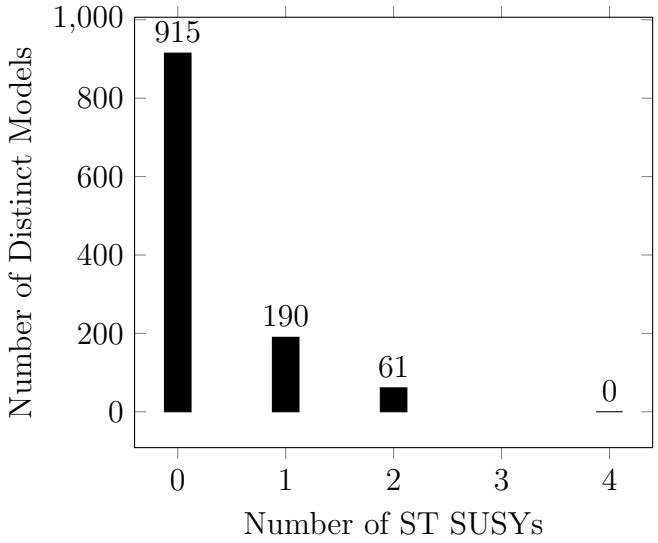


Figure 6.7: The number of ST SUSYs in the NAHE variation + O3L1 data set.

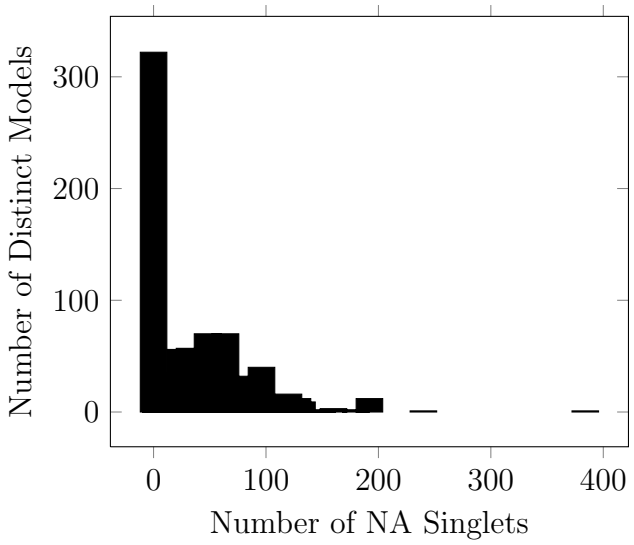


Figure 6.8: The number of non-Abelian singlets in the NAHE variation + O3L1 data set.

is more than one way to configure an observable sector, each configuration will be counted when tallying the charged exotics and net chiral generations.

6.4.1 E_6

The equation for calculating the number of net chiral generations is given in equation 5.1. There were 101 unique models containing E_6 in the O2L1 data set, while the O3L1 data set had only 68 unique models. The hidden sector gauge group content for the O2L1 models is tabulated in Table 6.7. The hidden sector gauge groups for the O3L1 models is presented in Table 6.8. The most noticeable feature of both tables is that all of the models have $U(1)$ gauge groups accompanying the E_6. Most do not have the $SO(22)$ of the NAHE variation, however. Statistics related to the number of net chiral fermion generations are presented in Figure 6.9 for the O2L1 data set, along with the number of charged exotics. All of the models in the O3L1 data set had no net chiral fermion generations or charged exotics. A vast majority of the models in the O2L1 data set do not have observable sector charged exotics or net chiral fermion generations, implying that the added basis vectors do not often alter the matter content of the models. Statistics for the number of gauge group factors, number of $U(1)$ factors, the number of ST SUSYs, and the number of non-Abelian singlets are presented in Figure 6.10 for the O2L1 data set and Figure 6.11 for the O3L1 data set.

6.4.2 $SO(10)$

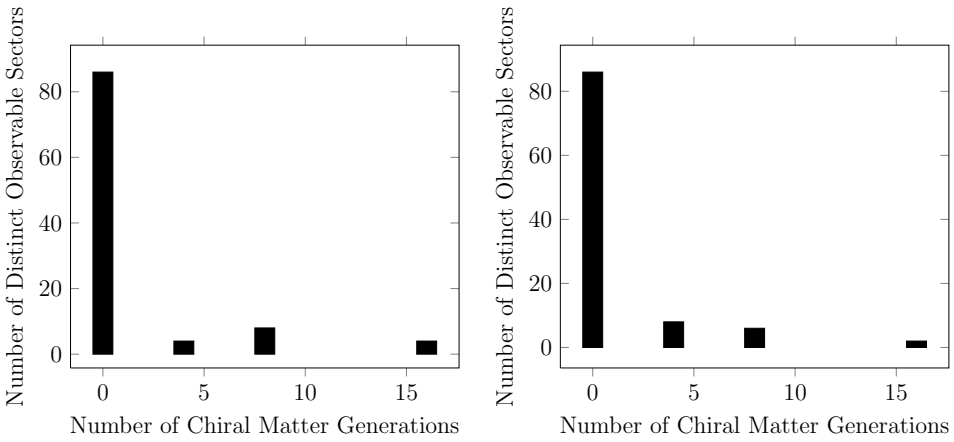
The number of net chiral fermion generations for the $SO(10)$ GUT group is given by equation (5.2). There were 125 models with $SO(10)$ in the O2L1 data set and 271 models with $SO(10)$ in the O3L1 data set. The hidden sector gauge content is presented in Table 6.9 for the O2L1 models and Table 6.10 for the O3L1 models. Note that in both data sets all of the models with $SO(10)$ also come with a $U(1)$ gauge group, implying the mechanism for reducing $SO(22)$ (or more

Table 6.7: The hidden sector gauge group content for the NAHE variation + O2L1 E_6 models.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	14	13.86%
$SU(2)^{(2)}$	8	7.921%
$SU(4)$	10	9.901%
$SO(5)$	6	5.941%
$SO(7)$	2	1.98%
$SO(9)$	6	5.941%
$SO(11)$	6	5.941%
$SO(13)$	6	5.941%
$SO(15)$	2	1.98%
$SO(17)$	6	5.941%
$SO(19)$	8	7.921%
$SO(21)$	6	5.941%
$SO(8)$	8	7.921%
$SO(10)$	14	13.86%
$SO(12)$	14	13.86%
$SO(14)$	9	8.911%
$SO(16)$	9	8.911%
$SO(18)$	14	13.86%
$SO(20)$	12	11.88%
$SO(22)$	8	7.921%
E_8	1	0.9901%
$U(1)$	101	100%

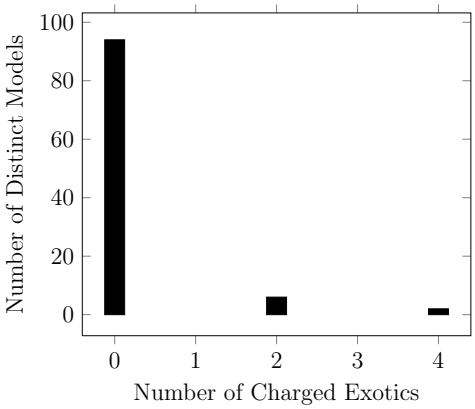
Table 6.8: The hidden sector gauge groups for the NAHE variation + O3L1 E_6 models.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	31	45.59%
$SU(3)$	1	1.471%
$SU(4)$	12	17.65%
$SU(6)$	4	5.882%
$SU(8)$	6	8.824%
$SU(9)$	10	14.71%
$SU(10)$	8	11.76%
$SU(11)$	5	7.353%
$SU(12)$	4	5.882%
$SU(13)$	1	1.471%
$SO(8)$	9	13.24%
$SO(10)$	15	22.06%
$SO(12)$	12	17.65%
$SO(14)$	5	7.353%
$SO(16)$	3	4.412%
$SO(18)$	4	5.882%
$SO(20)$	2	2.941%
$SO(22)$	2	2.941%
E_8	2	2.941%
$U(1)$	68	100%



(a) With Hidden Sector Duplicates

(b) Without Hidden Sector Duplicates



(c)

Figure 6.9: The number of chiral matter generations and charged exotics for E_6 models in the NAHE variation + O2L1 data set.

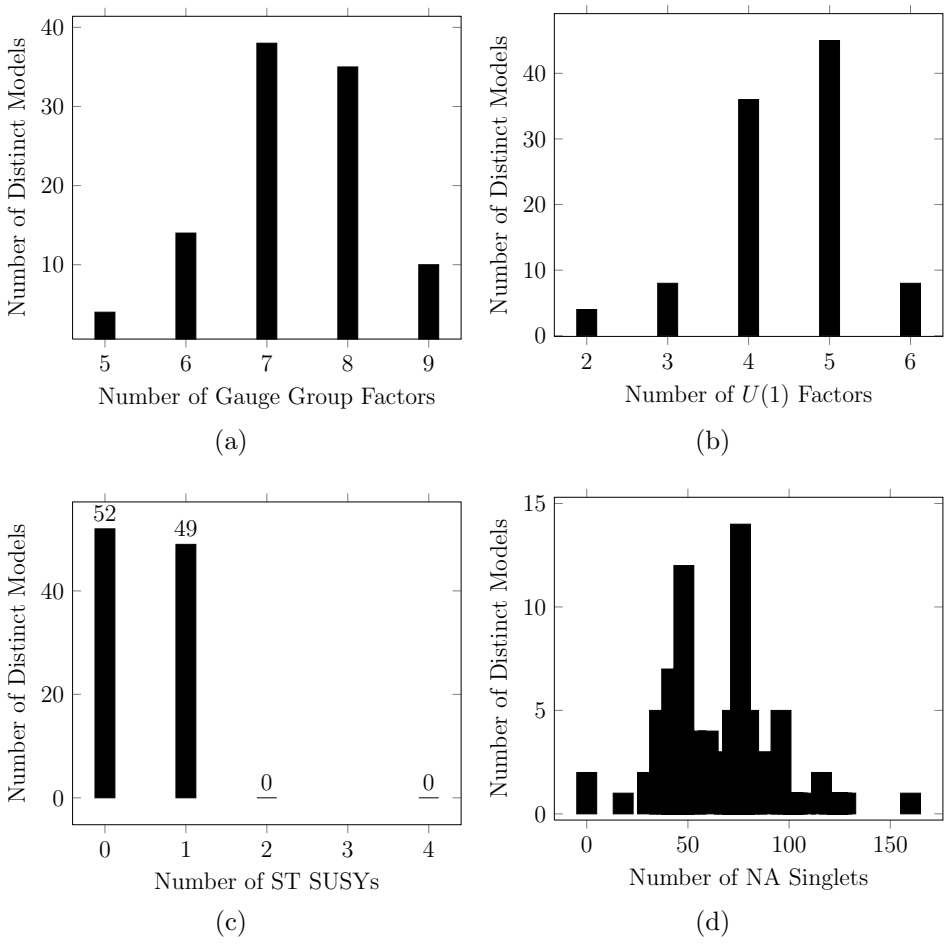


Figure 6.10: Statistics for the E_6 models in the NAHE variation + O2L1 data set.

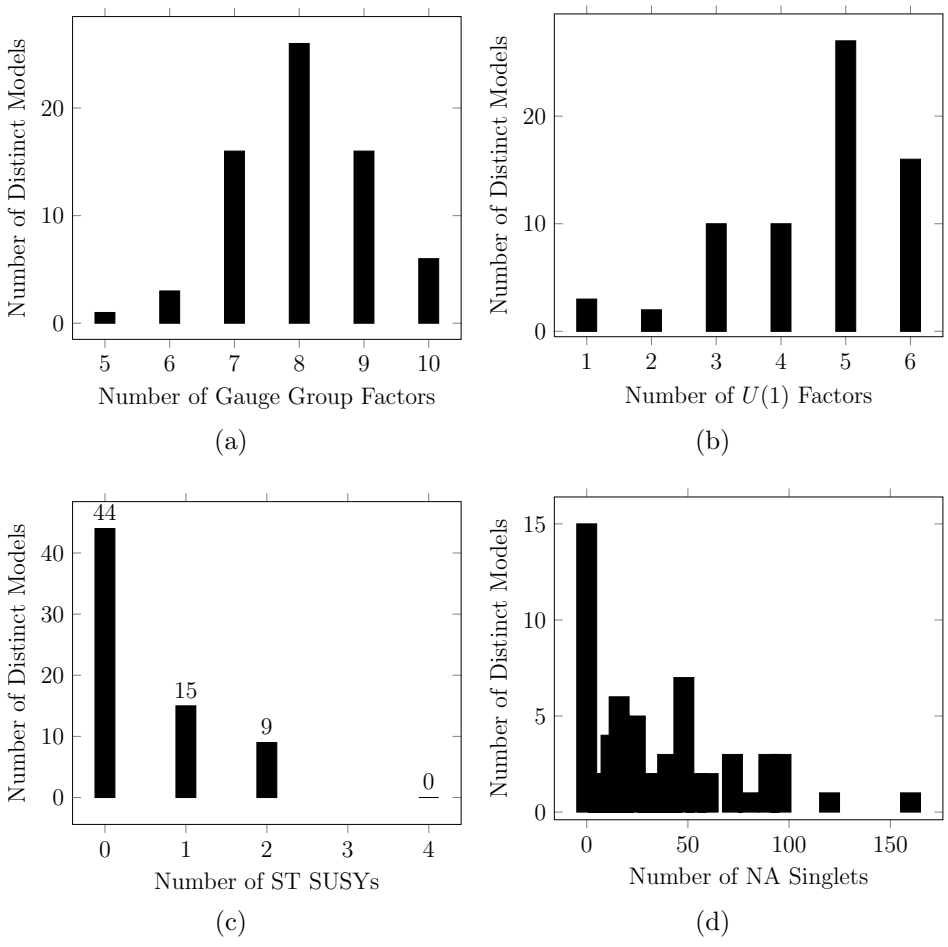


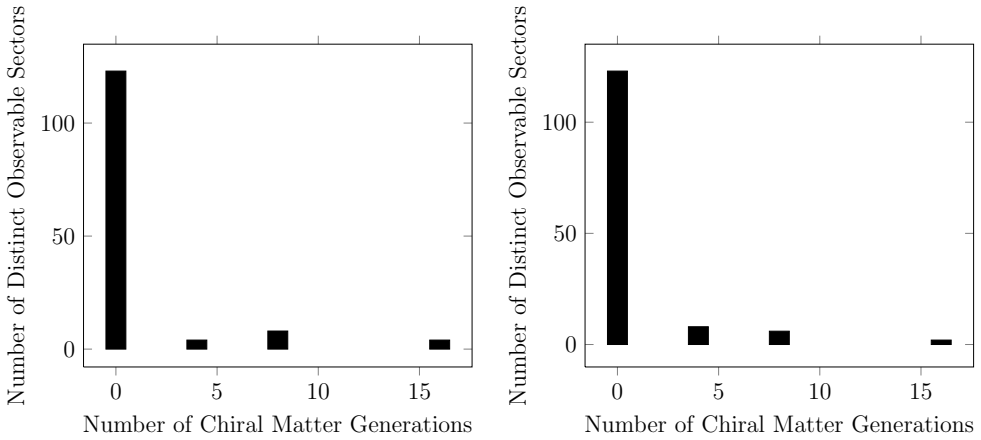
Figure 6.11: Statistics for the E_6 models in the NAHE variation + O3L1 data set.

Table 6.9: Hidden sector gauge content of the NAHE variation + O2L1 $SO(10)$ models.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	23	18.4%
$SU(2)^{(2)}$	8	6.4%
$SU(4)$	10	8%
$SU(6)$	10	8%
$SO(5)$	6	4.8%
$SO(7)$	2	1.6%
$SO(9)$	6	4.8%
$SO(11)$	6	4.8%
$SO(13)$	6	4.8%
$SO(15)$	2	1.6%
$SO(17)$	6	4.8%
$SO(19)$	8	6.4%
$SO(21)$	6	4.8%
$SO(8)$	8	6.4%
$SO(12)$	35	28%
$SO(14)$	10	8%
$SO(16)$	10	8%
$SO(18)$	14	11.2%
$SO(20)$	12	9.6%
$SO(22)$	9	7.2%
E_6	14	11.2%
E_7	1	0.8%
$U(1)$	125	100%

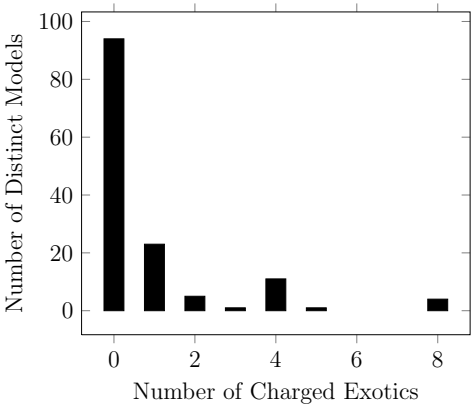
Table 6.10: Hidden sector gauge content of the NAHE variation + O3L1 $SO(10)$ models.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	155	57.2%
$SU(3)$	27	9.963%
$SU(4)$	59	21.77%
$SU(5)$	14	5.166%
$SU(6)$	59	21.77%
$SU(7)$	22	8.118%
$SU(8)$	24	8.856%
$SU(9)$	36	13.28%
$SU(10)$	26	9.594%
$SU(11)$	19	7.011%
$SU(12)$	11	4.059%
$SU(13)$	1	0.369%
$SU(14)$	1	0.369%
$SO(8)$	48	17.71%
$SO(12)$	35	12.92%
$SO(14)$	22	8.118%
$SO(16)$	10	3.69%
$SO(18)$	7	2.583%
$SO(20)$	2	0.738%
$SO(22)$	3	1.107%
E_6	15	5.535%
E_7	4	1.476%
E_8	2	0.738%
$U(1)$	271	100%

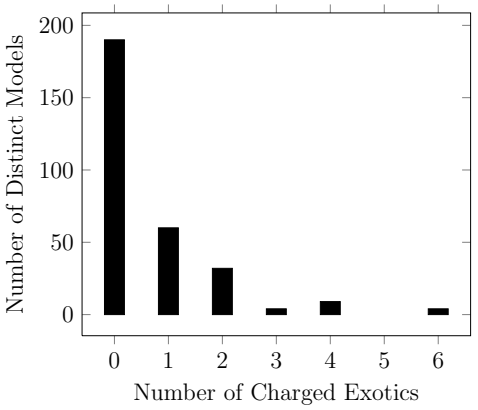


(a) O2L1 with hidden sector duplicates.

(b) O2L1 without hidden sector duplicates.



(c) O2L1.



(d) O3L1.

Figure 6.12: Statistics related to the observable sector in the $SO(10)$ NAHE variation models.

likely E_6) is not unifying the $U(1)$'s into a larger group. The number of net chiral fermion generations in the O2L1 data set with and without hidden sector duplicates is presented in Figure 6.12. No models in the O3L1 data set had any net chiral matter generations, but both data sets contained observable sector charged exotics. Those are also plotted in Figure 6.12. Since most models in the O2L1 data set and all of the models in the O3L1 data set had zero net chiral matter generations, it is clear that more complicated basis vector sets — either higher order or more layers — will be needed to produce quasi-realistic $SO(10)$ GUT models. It is also clear

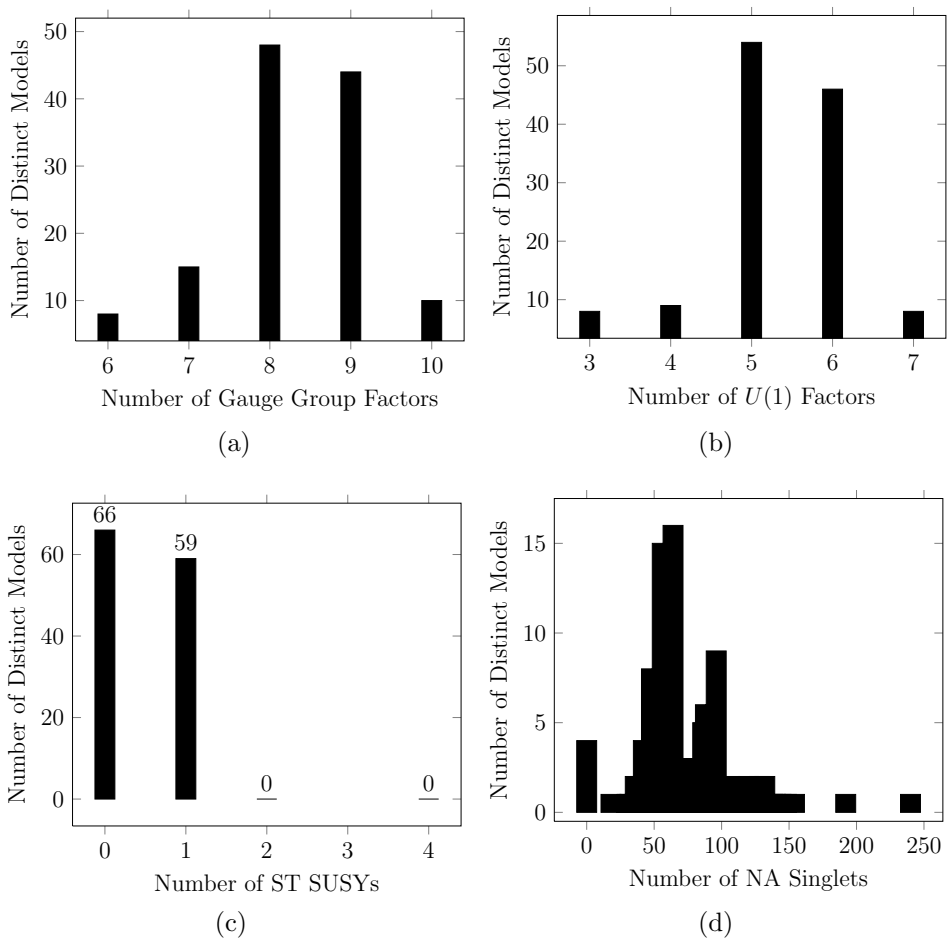


Figure 6.13: Statistics for the NAHE variation + O2L1 $SO(10)$ models.

that the number of observable sector charged exotics is peaked at zero. This would be advantageous if more models had a net number of chiral fermion generations. It is likely that the models without observable sector charged exotics do not have any net chiral matter generations, thus limiting the phenomenological advantages of having few exotics. The remaining statistics for the $SO(10)$ models are presented in Figure 6.13 for the O2L1 models and Figure 6.14 for the O3L1 models.

6.4.3 $SU(5) \otimes U(1)$

The number of net chiral matter generations in a flipped $SU(5)$ model is presented in equation (5.3). As was the case with the NAHE set, there were not

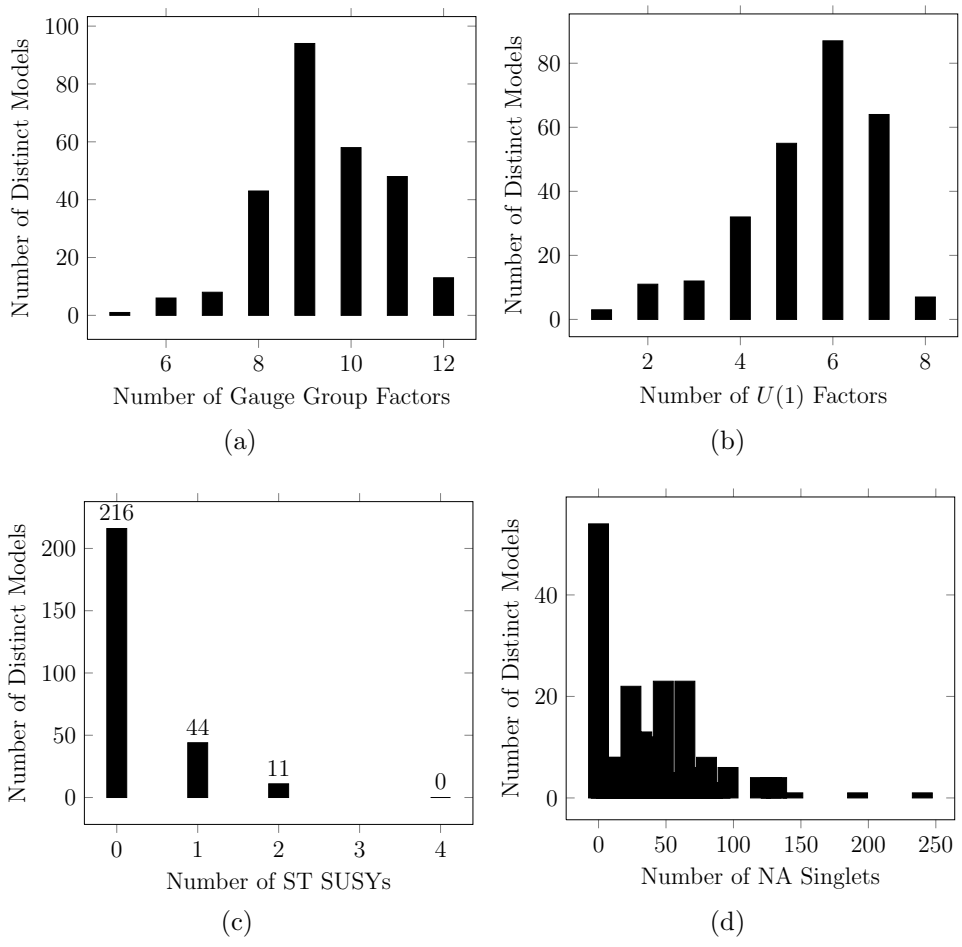


Figure 6.14: Statistics for the NAHE variation + O3L1 $SO(10)$ models.

Table 6.11: The hidden sector gauge group content of the $SU(5) \otimes U(1)$ models in the NAHE variation + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(2)$	87	52.73%
$SU(3)$	19	11.52%
$SU(4)$	28	16.97%
$SU(6)$	8	4.848%
$SU(7)$	20	12.12%
$SU(8)$	23	13.94%
$SU(9)$	34	20.61%
$SU(10)$	35	21.21%
$SU(11)$	32	19.39%
$SU(12)$	1	0.6061%
$SO(8)$	22	13.33%
$SO(10)$	14	8.485%
$SO(12)$	7	4.242%
$SO(14)$	5	3.03%

NAHE variation based O2L1 extensions producing an $SU(5) \otimes U(1)$ gauge group. The O3L1 extensions, however, produced 165 models containing $SU(5) \otimes U(1)$. The hidden sector gauge content of those models is presented in Table 6.11. Most of the models have the $SU(5)$ GUT group accompanied by another $SU(N+1)$ gauge group, a result of the models being built from an extension with an odd ordered RM. The ranks of these $SU(N+1)$ groups can get quite large. This is likely due to the additional twisted sector from the extension breaking the $SO(22)$ group of the NAHE variation into the $SU(N+1)$ groups. None of the models in this data set

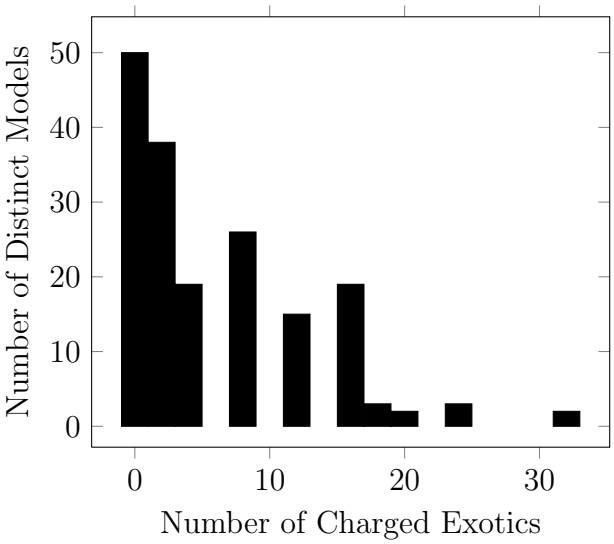


Figure 6.15: The number of observable sector charged exotics for the flipped-$SU(5) \otimes U(1)$ models in the NAHE variation + O3L1 data set.

had net chiral matter generations. The number of observable sector charged exotics is presented in Figure 6.15. Other statistics related to the $SU(5) \otimes U(1)$ models in the NAHE variation + O3L1 data set are presented in Figure 6.16.

6.4.4 Pati-Salam

The Pati-Salam gauge group is $SO(6) \otimes SO(4)$, which is isomorphic to $SU(4) \otimes SU(2) \otimes SU(2)$. The number of net chiral matter generations is given by equation (5.4). As was the case with the NAHE investigations, the generations of matter and anti-matter follow the same statistics, since all of the possible permutations of observable sectors are counted. There are no models in the O2L1 data set containing the Pati-Salam gauge group, but the O3L1 data set had 125 such models. The hidden sector gauge content of those models is presented in Table 6.12. The hidden sector gauge group appearing in the most models is $U(1)$, which is expected. Additional $U(1)$ groups are common with basis vectors of fractional phases, such as the ones in this data set. All of the models in this data set had zero net chiral matter generations, suggesting more complex basis vectors should be used to construct quasi-realistic

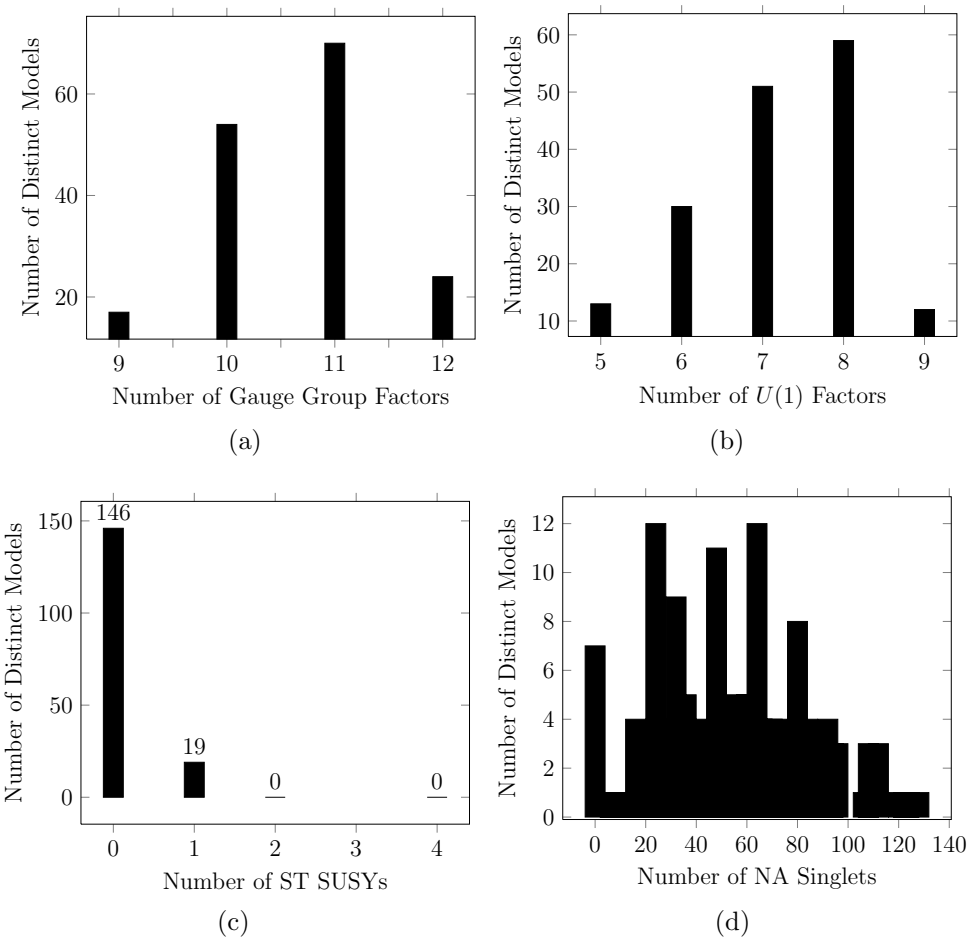


Figure 6.16: Statistics for the $SU(5) \otimes U(1)$ models in the NAHE variation + O3L1 data set.

Table 6.12: The hidden sector gauge group content for the Pati-Salam models in the NAHE variation + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(3)$	12	9.6%
$SU(5)$	8	6.4%
$SU(6)$	25	20%
$SU(8)$	29	23.2%
$SU(9)$	24	19.2%
$SU(10)$	15	12%
$SU(11)$	3	2.4%
$SU(12)$	7	5.6%
$SO(8)$	9	7.2%
$SO(10)$	11	8.8%
$SO(12)$	22	17.6%
$SO(14)$	19	15.2%
$SO(16)$	4	3.2%
$SO(20)$	2	1.6%
E_6	1	0.8%
$U(1)$	123	98.4%

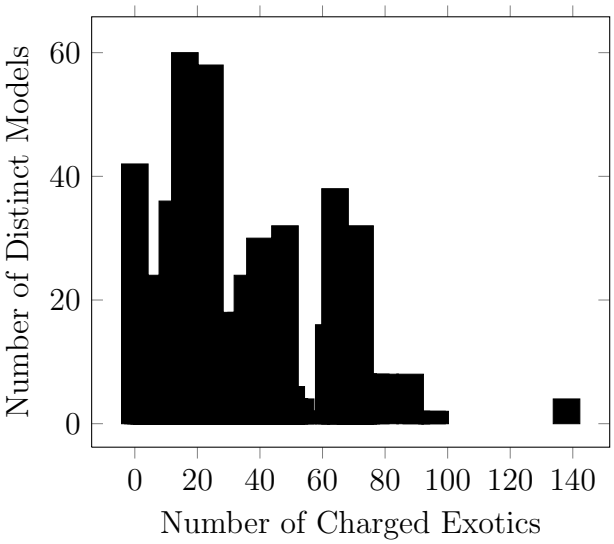


Figure 6.17: The number of observable sector charged exotics in the NAHE variation + O3L1 data set.

Pati-Salam models originating with this NAHE variation. The number of observable sector charged exotics is presented in Figure 6.17, while remaining statistics for these models is presented in Figure 6.18.

6.4.5 Left-Right Symmetric

The Left-Right Symmetric GUT group is a derivative of the Pati-Salam GUT, replacing the $SU(4)$ that governed lepton and quark generations to an $SU(3)$, which directly represents the QCD color force. As was the case with the NAHE extensions, only the quark generations will be statistically examined in this study. The number of net chiral quark generations is given by equation (5.5). There were no models containing this gauge groups in the O2L1 data set (none contained $SU(3)$), but there were 61 models in the O3L1 data set with this GUT group. The hidden sector gauge group content is presented in Table 6.13. Note that all of the models in this subset have $U(1)$ factors. This means each of them also contains an MSSM gauge group as well. Statistics on MSSM models will be presented in the next section.

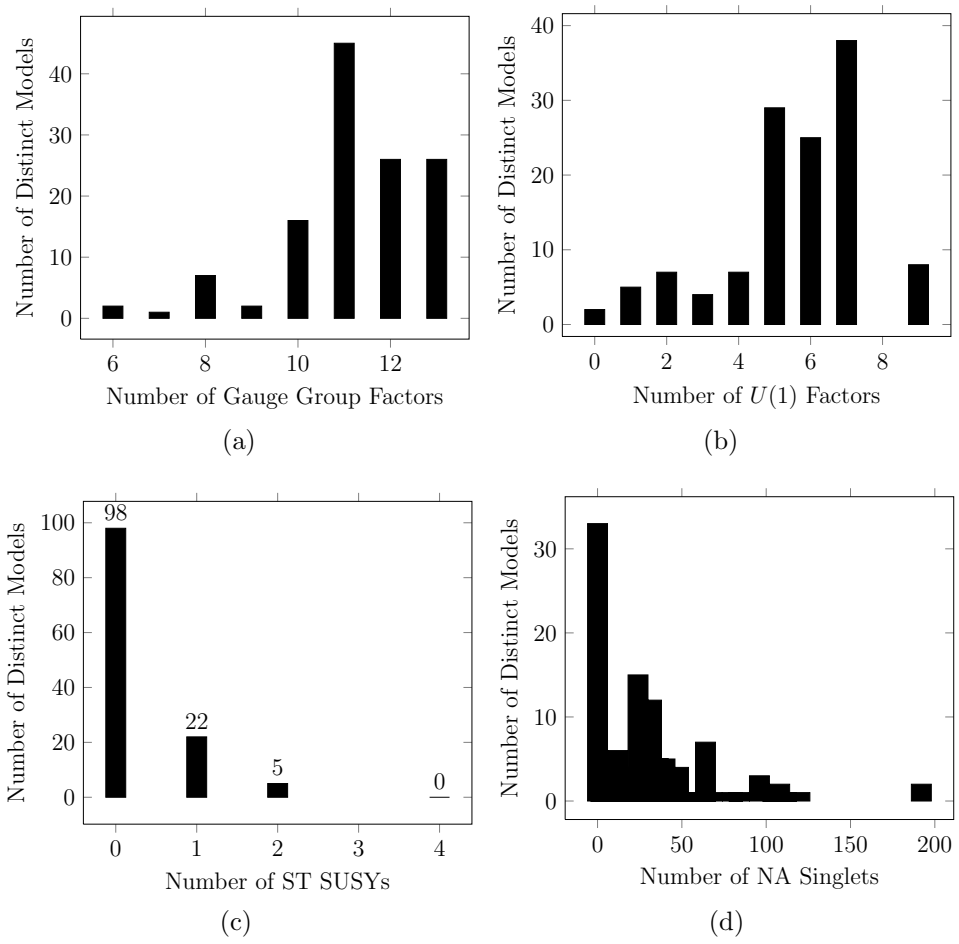


Figure 6.18: Statistics for the Pati-Salam models in the NAHE variation + O3L1 data set.

Table 6.13: The hidden sector gauge group content of the Left-Right Symmetric models in the NAHE variation + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(4)$	12	19.67%
$SU(7)$	14	22.95%
$SU(8)$	7	11.48%
$SU(9)$	9	14.75%
$SU(10)$	12	19.67%
$SU(11)$	17	27.87%
$SU(12)$	2	3.279%
$SO(8)$	8	13.11%
$SO(10)$	6	9.836%
$U(1)$	61	100%

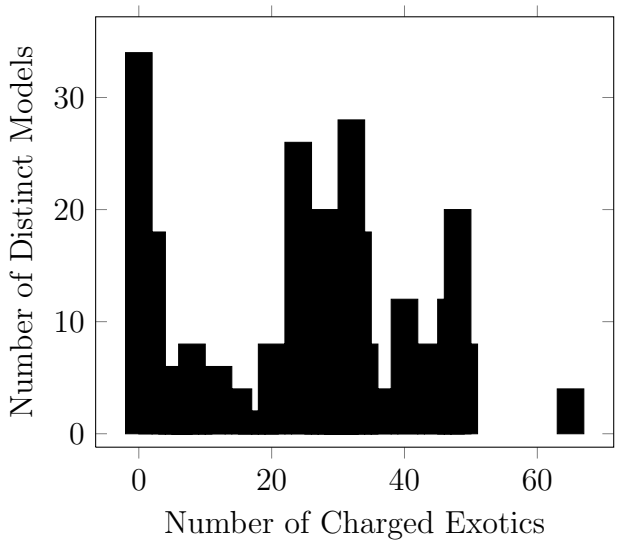


Figure 6.19: The number of observable sector charged exotics in the NAHE variation + O3L1 Left-Right Symmetric models.

As was the case with the Pati-Salam models, all of the models in this data set have zero net quark generations. The number of observable sector charged exotics is presented in Figure 6.19, and other statistics are presented in Figure 6.20.

6.4.6 MSSM-like Models

The MSSM[1] gauge group is $SU(3) \otimes SU(2) \otimes U(1)$. As with the NAHE investigation, only the quark generations will be statistically examined. Thus, the term chiral matter generation here refers only to quark generations. The equation for the number of net chiral matter generations is given by equation (5.6), while the number of net chiral anti-generations is given by equation (5.7). Due a lack of $SU(3)$ groups, no models in the NAHE variation + O2L1 data set contain the MSSM gauge groups. The O3L1 data set has 63 models with the MSSM group. The hidden sector gauge content of these models is presented in Table 6.14. A significant number of these models contain higher rank $SU(N+1)$ gauge groups, while not many contain

[1] As as the case with the NAHE investigation, MSSM here refers only to the gauge group. Models with this gauge group may or may not have ST SUSY.

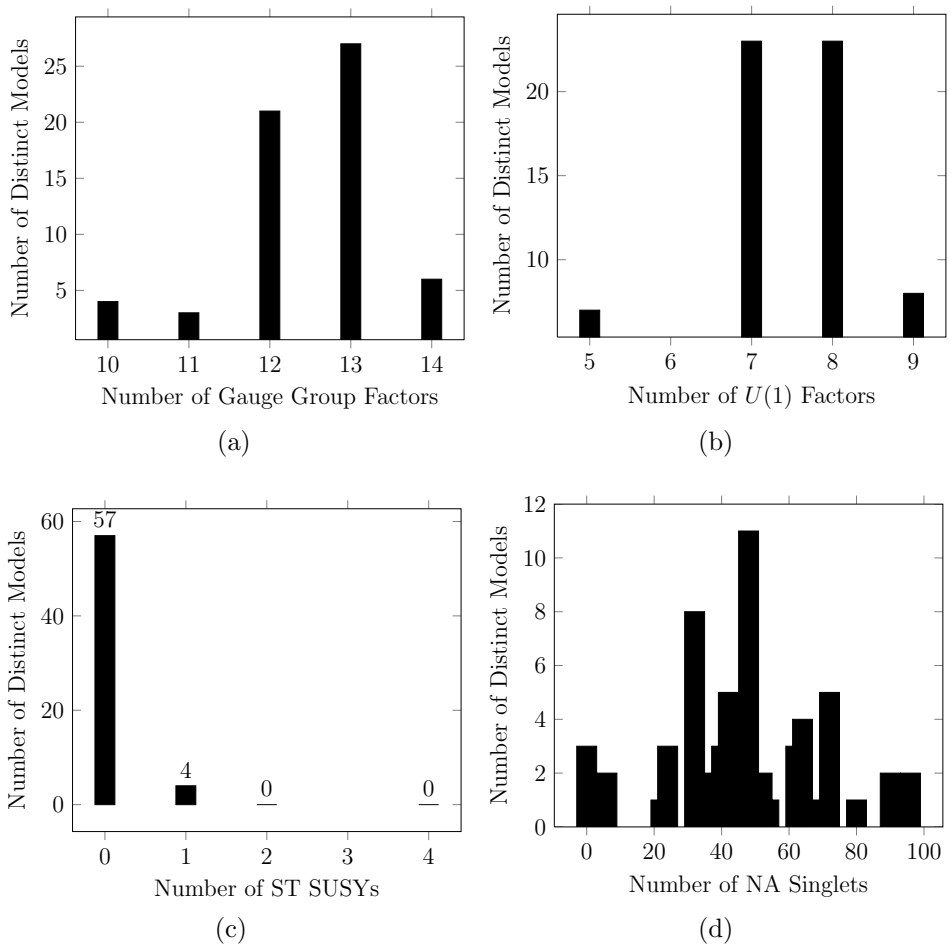


Figure 6.20: Statistics for the Left-Right Symmetric models in the NAHE variation + O3L1 data set.

Table 6.14: The hidden sector gauge groups for the MSSM models in the NAHE variation + O3L1 data set.

Gauge Group	Number of Unique Models	% of Unique Models
$SU(4)$	13	20.63%
$SU(6)$	1	1.587%
$SU(7)$	14	22.22%
$SU(8)$	7	11.11%
$SU(9)$	9	14.29%
$SU(10)$	12	19.05%
$SU(11)$	18	28.57%
$SU(12)$	3	4.762%
$SO(8)$	8	12.7%
$SO(10)$	6	9.524%

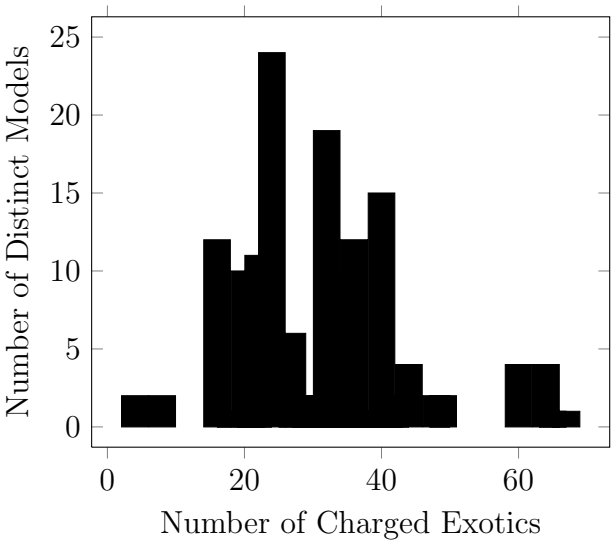


Figure 6.21: The number of observable sector charged exotics in the NAHE variation + O3L1 data set.

higher ranking $SO(2N)$ groups. None of them contain a group from the original NAHE variation. More complicated basis vector sets will be needed to see if the E_6 group in the NAHE variation can be broken to the MSSM without breaking the $SO(22)$. None of the possible observable sector choices yielded net chiral matter generations either. The number of observable sector charged exotics is presented in Figure 6.21, while other statistics related to these models are presented in Figure 6.22.

6.4.7 ST SUSYs

The ST SUSY distributions across the GUT group subsets was examined in the NAHE set investigations. It will be examined here as well. The distributions of ST SUSYs for the full NAHE variation + O2L1 data set, the O2L1 E_6 models, and the O2L1 $SO(10)$ models are plotted in Figure 6.23. The distributions of ST SUSYs for the full NAHE variation + O3L1 data set, the E_6 models, the $SO(10)$ models, the $SU(5) \otimes U(1)$ models, and the Pati-Salam models in the O3L1 data set are plotted in Figure 6.24. The distributions of ST SUSYs for the full NAHE

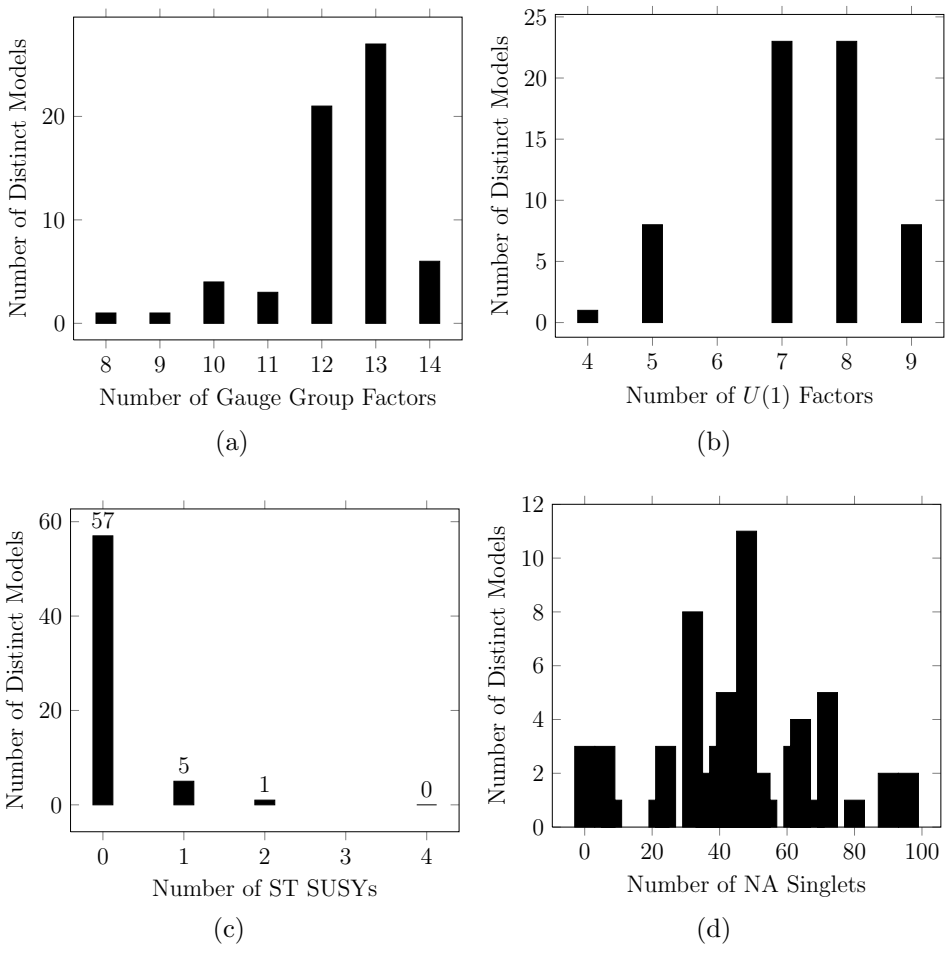
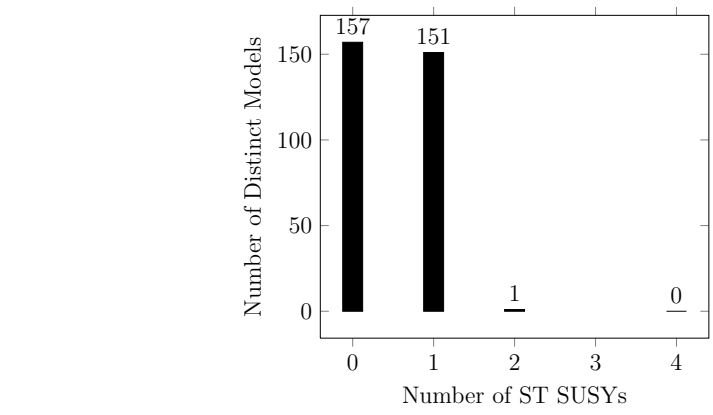
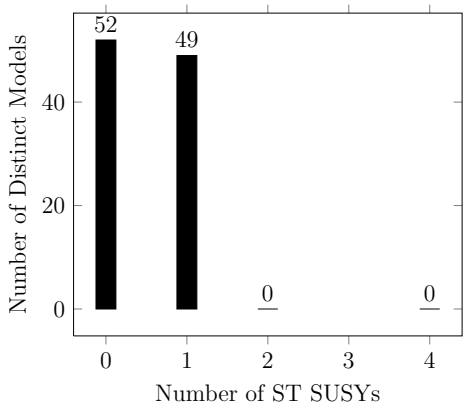


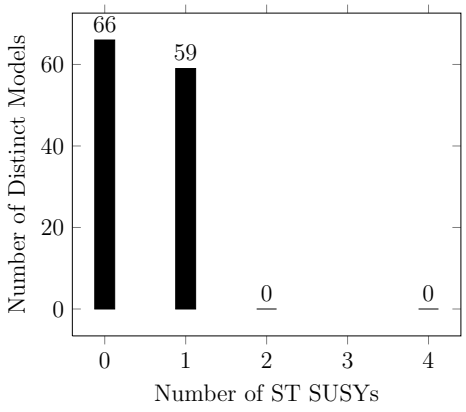
Figure 6.22: Statistics for the MSSM models in the NAHE variation + O3L1 data set.



(a) Full data set.

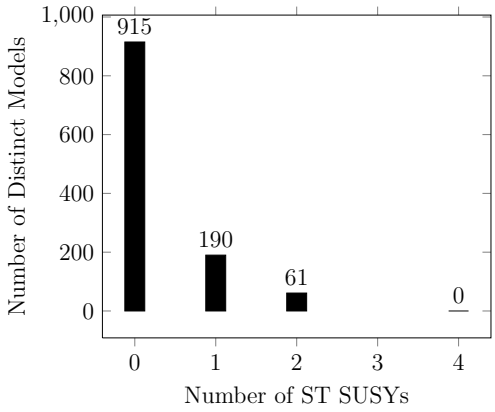


(b) E_6 Models.

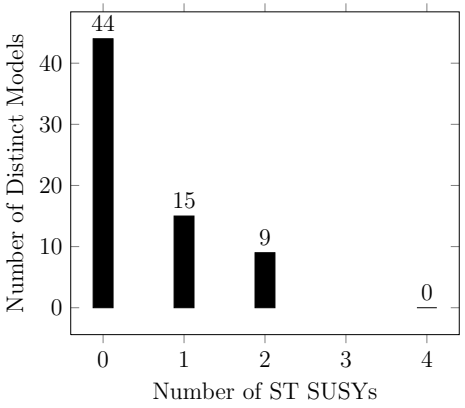


(c) $SO(10)$ Models.

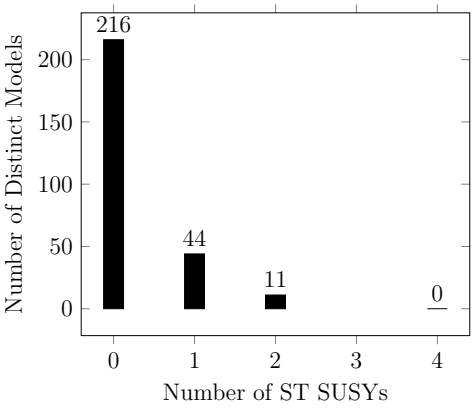
Figure 6.23: The distributions of ST SUSYs for the NAHE variation + O2L1 GUT group data sets.



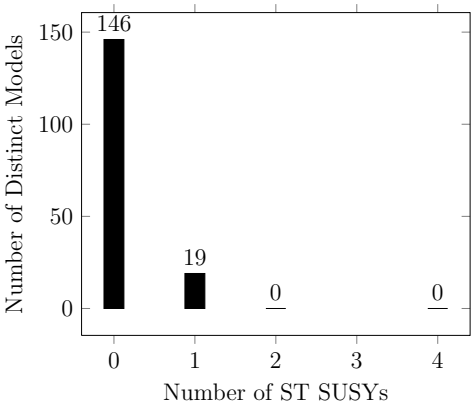
(a) Full data set.



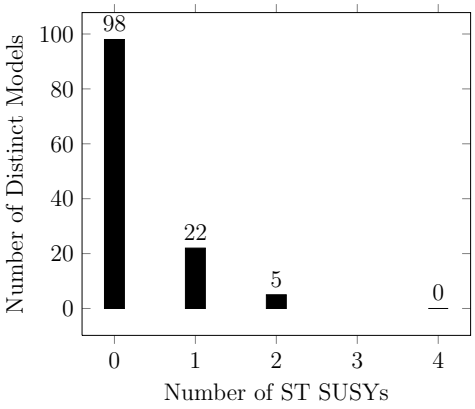
(b) E_6 Models.



(c) $SO(10)$ models.

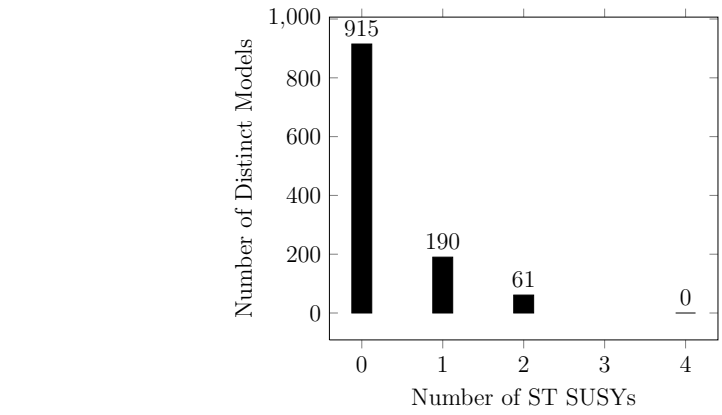


(d) $SU(5) \otimes U(1)$ models.

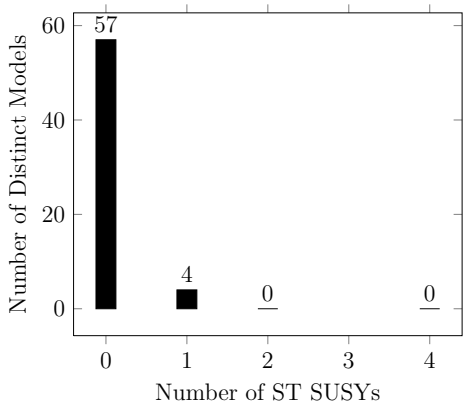


(e) Pati-Salam models.

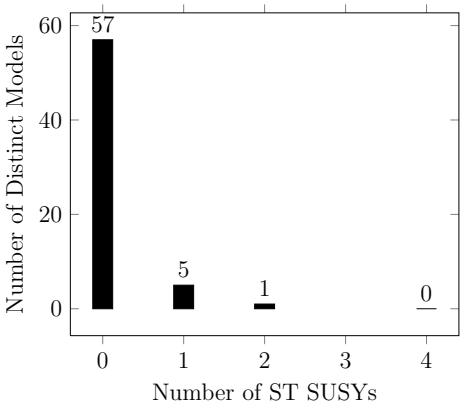
Figure 6.24: The distributions of ST SUSYs for the NAHE variation + O3L1 GUT group data sets.



(a) Full data set.



(b) Left-Right Symmetric models.



(c) MSSM models.

Figure 6.25: The distributions of ST SUSYs for the NAHE variation + O3L1 GUT group data sets.

variation + O3L1 data set, the Left-Right Symmetric models, and the MSSM models in the O3L1 data set are plotted in Figure 6.25. The O2L1 models all have the same distributions regardless of which GUT is chosen. In these models, the gauge content does not statistically couple to the ST SUSY. For the O3L1 models, however, some of the GUT groups do appear to couple with the ST SUSY. In particular, the E_6 models have a greater percentage of models with $N = 2$ ST SUSY. The $SU(5)\otimes U(1)$ modes, Left-Right Symmetric models, and the MSSM models have a lower percentage of models with $N = 1$ ST SUSY as well. As all of the models containing these GUTs

Table 6.15: The basis vector and k_{ij} row of the NAHE variation extension producing the $SO(11) \otimes SO(11) \otimes SO(10) \otimes U(1)^5$ mirrored model.

Sec	O	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,...,5}$	$\overline{\eta}^1$	$\overline{\eta}^2$	$\overline{\eta}^3$	$\overline{\phi}^{1,...,8}$
$\vec{v}$	2	1	1	1	1	1,...,1	1,...,1	0,...,0,1,1,1		

Sec	O	$y^{12}\|\|\overline{y}^{12}$	$y^{34}\|\|\overline{y}^{34}$	$y^{56}\|\|\overline{y}^{56}$	$w^{1,...,6}\|\|\overline{w}^{1,...,6}$
$\vec{v}$	2	0,0||0,0	1,1||1,1	0,1||0,1	0,0,1,1,0,1||0,1,1,1,1,1

$$k_{\vec{v},j} = (1,1,1,1,1)$$

have at least a single $U(1)$, there could be a correlation between the number of $U(1)$'s and the number of ST SUSYs.

6.5 Models with Mirroring

The larger sets of matching boundary conditions, seen in Table 6.1, are expected to lead to models with mirrored gauge groups and matter states. Only one model in those discussed thus far exhibit full gauge mirroring, and the matter states are not mirrored. Rather than examine that model, three models that appeared in the first NAHE variation study [52] that have two mirrored gauge groups with a single "shadow" gauge group will be discussed. They have mostly mirrored matter representations as well, with limited coupling between the mirrored gauge groups and the shadow group.

The first of these models is an order-2 extension to the NAHE variation with a gauge groups $SO(11) \otimes SO(11) \otimes SO(10) \otimes U(1)^5$. The basis vector producing this model are presented in Table 6.15, and the particle content is presented in Table 6.16. The matter representations of the $SO(11)$ gauge groups are completely mirrored, but not decoupled. There is one (11,11,1) representation that couples the representations of the two $SO(11)$ groups. Neither of the $SO(11)$ groups have a representation charged under the shadow $SO(10)$, however. Analysis of the $U(1)$

Table 6.16: The particle content of the $SO(11) \otimes SO(11) \otimes SO(10) \otimes U(1)^5$ mirrored model. The model has $N = 0$ ST SUSY.

QTY	$SO(11)$	$SO(11)$	$SO(10)$
2	32	1	1
1	11	11	1
16	11	1	1
2	1	32	1
16	1	11	1
8	1	1	$\overline{16}$
12	1	1	10
88	1	1	1
12	1	1	16

charges will determine how "deep" the mirroring goes into the phenomenology. A C++ class for $U(1)$ analysis is currently being written, and will be implemented soon.

Another model that shows a similar mirroring with a shadow gauge group with gauge group $SO(10) \otimes SO(10) \otimes SO(14) \otimes U(1)^5$ will be presented. The basis vector making up this model are tabulated in Table 6.17, and the particle content is tabulated in Table 6.18. This model is an order-3 model constructed from the NAHE variation, but without the $\vec{S}$ sector generating the ST SUSY. The model generates its own gravitino sector via the mechanism described in the $D = 10$ study. In this model, the mirrored $SO(10)$'s do not couple to one another, but each has a single (10,14) state that couples to the shadow $SO(14)$. As was the case with the mirrored $SO(11)$ model, this model will be a good candidate for $U(1)$ analysis.

The final mirrored model presented in this study has a gauge group $E_6 \otimes E_6 \otimes SO(14) \otimes U(1)$. The basis vector producing this model is presented in Table 6.19, and the particle content of this model is presented in Table 6.20. Each of the gauge groups in this model is completely decoupled from the others. It is likely this mirroring is more common amongst models with larger mirrored gauge groups;

Table 6.17: The basis vector and k_{ij} row of the NAHE variation extension producing the $SO(10) \otimes SO(10) \otimes SO(14) \otimes U(1)^5$ mirrored model.

Sec	O	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,...,5}$	$\overline{\eta}^1$	$\overline{\eta}^2$	$\overline{\eta}^3$	$\overline{\phi}^{1,...,8}$
$\vec{v}$	3	1	1	1	1	0,...,0	0	$\frac{2}{3}$	$\frac{2}{3}$	0,...,0,$\frac{2}{3}$,$\frac{2}{3}$

Sec	O	$y^{12}\|\|\overline{y}^{12}$	$y^{34}\|\|\overline{y}^{34}$	$y^{56}\|\|\overline{y}^{56}$	$w^{1,...,6}\|\|\overline{w}^{1,...,6}$
$\vec{v}$	3	0,0||$\frac{2}{3}$,$\frac{2}{3}$	0,0||$\frac{2}{3}$,$\frac{2}{3}$	0,0||$\frac{2}{3}$,$\frac{2}{3}$	0,0,0,0,0,0||$\frac{2}{3}$,0,$\frac{2}{3}$,$\frac{2}{3}$,0,$\frac{2}{3}$

$$k_{\vec{v},j} = (1,1,1,1)$$

Table 6.18: The particle content of the $SO(10) \otimes SO(10) \otimes SO(14) \otimes U(1)^5$ model. This model has $N = 0$ ST SUSY.

QTY	$SO(10)$	$SO(10)$	$SO(14)$
8	16	1	1
1	10	1	14
14	10	1	1
8	1	16	1
1	1	10	14
14	1	10	1
12	1	1	14
40	1	1	1
8	1	$\overline{16}$	1
8	$\overline{16}$	1	1

Table 6.19: The basis vector and k_{ij} row of the NAHE variation extension producing the $E_6 \otimes E_6 \otimes SO(14) \otimes U(1)^3$ mirrored model.

Sec	O	ψ	x^{12}	x^{34}	x^{56}	$\overline{\psi}^{1,...,5}$	$\overline{\eta}^1$	$\overline{\eta}^2$	$\overline{\eta}^3$	$\overline{\phi}^{1,...,8}$
$\vec{v}$	3	1	1	0	0	$\frac{2}{3}$,...,$\frac{2}{3}$	0	0	$\frac{2}{3}$	0,0,0,$\frac{2}{3}$,...,$\frac{2}{3}$

Sec	O	$y^{12}\|\|\overline{y}^{12}$	$y^{34}\|\|\overline{y}^{34}$	$y^{56}\|\|\overline{y}^{56}$	$w^{1,...,6}\|\|\overline{w}^{1,...,6}$
$\vec{v}$	3	0,0||0,0	1,1||0,0	1,1||$\frac{2}{3}$,$\frac{2}{3}$	0,0,0,0,0,0||0,0,0,0,0,0

$$k_{\vec{v},j} = (0,1,0,1,0)$$

Table 6.20: The particle content of the $E_6 \otimes E_6 \otimes SO(14) \otimes U(1)^3$ model. This model has $N = 2$ ST SUSY.

QTY	$SO(14)$	E_6	E_6
12	14	1	1
6	1	27	1
6	1	1	27
6	1	1	$\overline{27}$
6	1	$\overline{27}$	1

the masslessness conditions for the fermion states constrain the total dimensions of transformation under the gauge groups.

While none of these models can serve as a quasi-realistic mirrored model, they do serve to highlight the features of the NAHE variation that are conducive to observable-hidden sector mirroring. Steps will be taken in future analysis to further automate searches for mirrored models with a shadow sector.

6.6 Conclusions

Though there were many models containing GUTs in the data sets explored in this study, a vast majority of them do not contain any net chiral fermion generations. No three-generation models were found. These conculsions are summarized in Table **??**. While there were more models with GUT gauge groups in the NAHE variation + O3L1 data set, none of them had any net chiral matter generations, implying that the added basis vector produces the barred and unbarred generations in even pairs, if at all. More complicated basis vector sets will need to be studied to see if any NAHE variation based quasi-realistic models can be constructed.

The distributions of ST SUSYs across the subsets of GUT models was also examined. It was concluded that, as was the case with the NAHE study, E_6 has a statistical coupling to enhanced ST SUSYs for order-3 models. Additionally, data

Table 6.21: A summary of the GUT group study with regard to the number of chiral fermion generations in the NAHE variation investigation.

GUT	Net Chiral Generations?	Three Generations?
O2L1 E_6	Yes	No
O2L1 $SO(10)$	Yes	No
O3L1 E_6	No	No
O3L1 $SO(10)$	No	No
O3L1 $SU(5) \otimes U(1)$	No	No
O3L1 Pati-Salam	No	No
O3L1 L-R Symmetric	No	No
O3L1 MSSM	No	No

sets in which all of the models contained at least one $U(1)$ factor had fewer models with $N = 1$ ST SUSY.

Models with partial gauge group mirroring were also discussed, with three cases presented: one in which the representations of the mirrored groups coupled to one another, one in which they coupled with a third "shadow" gauge group, and one in which they were completely decoupled. While a statistical search algorithm for finding quasi-mirrored models has not yet been completed, it will be used in future work to examine models with this property statistically.

APPENDICES

APPENDIX A

FF Framework Documentation

A.1 Introduction

Included here is the documentation for the FF Framework, a collection of C++ classes which serve as a tool set for constructing FFHS models. Details are presented on each class regarding members and usage, with more difficult or complicated algorithms detailed explicitly as needed. Detailed descriptions of class inheritances are also included, as well as instructions for operating the makefile.

A.2 FF Framework Classes

A.2.1 Format of class information

The formatting for relevant information is printed as follows:

NAME: The name of the class

PURPOSE: What the class does within the FF Framework.

OBJECTS CREATED BY: Which classes in the framework create objects of this type.

USED IN: Which classes in the framework use objects of this type.

MEMBERS: Descriptions of each of the members in the class.

CONSTRUCTORS: Descriptions of each of the constructors for the class. Copy constructors are not included.

METHODS: Descriptions of each of the methods in the class.

In addition, each class will have an accessor and a setter for each member. Accessors are **const** functions which return by value for fundamental C++ types and return by

const reference for C++ STL and FF Framework types. Setters are **void** functions which take an argument of the same type as the accessor of the member which is being set. Accessors are named the same as the member without the tailing underscore, while setters are named with **Set_** appended to the front of the member name without the tailing underscore. Each class has a copy constructor, which takes as an argument a **const** reference to an object of the same class and copies all members of the new class onto ***this**. Each class also has a destructor which is empty, allowing STL container destructors to handle memory deallocation.

A.2.2 FF_Alpha.hh

NAME: Alpha

PURPOSE: Serves as a bridge between the Basis_Alpha class and the Alpha_Fermion, Alpha_Boson, and Alpha_SUSY classes. It is used before the type and mass of the alpha have been determined by the Model_Builder class.

OBJECTS CREATED BY: Alpha_Builder

USED IN: Model_Builder, Alpha_Builder

MEMBERS:

vector¡int¿ Coefficients_ The coefficients which produced ***this** in Alpha_Builder.

CONSTRUCTORS:

Alpha() A default constructor. Does not initialize members.

Alpha(const vector¡int>& Numerator, int Denominator,

const vector<int>& Coefficients) Initializes members of the same name, some of which are in the base class, whose constructor is also called.

METHODS:

int Mass_Left() Returns the mass squared of the left moving part of ***this**.

int Mass_Right() Returns the mass squared of the right moving part of ***this**.

virtual char Type() const Returns the character type of ***this** when called through one of the inherited classes. For the **Alpha** class it returns '**n**'.

bool operator<(const Alpha& Other_Alpha) const Compares the numerators using the STL vector < operator.

void Display_Coefficients() const Prints the **Coefficients_** member onto the screen for debugging purposes.

A.2.3 FF_Alpha_Boson.hh

NAME: Alpha_Boson

PURPOSE: Holds the data for a boson sector. It also has a character member which indicates which subclass of **Alpha** is being passed to **State_Builder**.

OBJECTS CREATED BY: Alpha_Builder

USED IN: Model_Builder, State_Builder, Alpha_Builder

MEMBERS: None.

CONSTRUCTORS:

Alpha_Boson(const vector<int>& Numerator, int Denominator,

const vector<int>& Coefficients) All arguments are passed to the **Alpha** class's constructor.

METHODS:

char Type() const Returns the character '**b**'. Used to determine which states to build in **State_Builder**.

A.2.4 FF_Alpha_Builder.hh

NAME: Alpha_Builder

PURPOSE: Builds the linear combinations of alphas from a set of basis alphas with matching denominators, checks whether or not they can produce massless states, then classifies them as Alpha_Boson, Alpha_Fermion, or Alpha_SUSY. It also contains a boolean flag in the event that not all basis alphas are linearly independent.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

vector<Basis_Alpha> Common_Basis_Alphas_ Used as a basis to build the AlphaS.

vector<int> Coefficient_Limits_ Holds the limits for the coefficients used to produce the linear combinations of Common_Basis_Alphas_.

set<Alpha_Boson> Alpha_Bosons_ Holds the boson sectors produced by Common_Basis_Alphas_ which are capable of producing massless states. The container is an STL set rather than a vector or list because the recursive algorithm used to build the linear combinations produces duplicates. This is something which could be improved in future revisions.

set<Alpha_Fermion> Alpha_Fermions_ Holds the fermion sectors produced by Common_Basis_Alphas_ which are capable of producing massless states. The container is an STL set rather than a vector or list because the recursive algorithm used to build the linear combinations produces duplicates. This is something which could be improved in future revisions.

set<Alpha_SUSY> Alpha_SUSYs_ Holds the SUSY sectors produced by **Common_Basis_Alphas_** which are capable of producing massless gravitino states. The container is an STL **set** rather than a **vector** or **list** because the recursive algorithm used to build the linear combinations produces duplicates. This is something which could be improved in future revisions.

bool Linearly_Independent_Alphas_ A boolean flag which is set to **false** if the basis alphas are not linearly independent.

See the **bool Linearly_Independent_Alpha(const Alpha& Last_Alpha)** method for more details.

CONSTRUCTORS:

Alpha_Builder(const vector<Basis_Alpha>& Common_Basis_Alphas,

const vector<Basis_Alpha>& Basis_Alphas) This constructor first initializes the **Common_Basis_Alphas_** member with the value of **Common_Basis_Alphas**. It then sets the **Coefficient_Limits_** member to the return value of the **Get_Coefficient_Limits(const vector<Basis_Alpha>& Basis_Alphas)** method. It also initializes the member **Linearly_Independent_Alphas_** to a default value of **true**.

METHODS:

void Build_Alphas() This method serves as an interface for the class. Firstly, it initializes an empty **Alpha** object with numerator, coefficients, and denominator equal to zero. It then begins the recursion by calling the **Add_Alphas(int Layer, Alpha Last_Alpha)** method with **Layer = 0** and the empty **Alpha** object.

void Display_Common_Basis_Alphas() const Prints the **Common_Basis_Alphas_** member onto the screen for debugging purposes.

void Display_Coefficient_Limits() const Prints the **Coefficient_Limits_** member onto the screen for debugging purposes.

void Display_Alpha_Bosons() const Prints the **Alpha_Bosons_** member onto the screen for debugging purposes.

void Display_Alpha_Fermions() const Prints the **Alpha_Fermions_** member onto the screen for debugging purposes.

void Display_Alpha_SUSYs() const Prints the **Alpha_SUSYs_** member onto the screen for debugging purposes.

void Display_All_Alphas() const Calls **Display_Alpha_Bosons()**, **Display_Alpha_Fermions()**, and **Display_Alpha_SUSYs()** functions sequentially for debugging purposes.

vector<int> Get_Coefficients(const vector<Basis_Alpha>&

Basis_Alphas) This method, called in the constructor, takes **Basis_Alphas** as an argument and extracts the denominator from each **Basis_Alpha** object in the container. Those denominators serve as limits on the possible coefficients that can be used when producing the linear combinations of **Common_Basis_Alphas_**. The denominators are pushed onto the vector **Coefficient_Limits** and returned.

void Add_Alphas(int Layer, Alpha Last_Alpha) A recursive method which actually builds the sectors. It essentially nests **for** loops (representing the coefficients which build the sectors) for each basis alpha in the model, then categorizes each result based on the type of **Alpha** object which is created. The method first checks if the variable **Layer** is less than the size of **Common_Basis_Alphas_**. If this statement evaluates to **true**, then the builder has not yet nested all of the **for** loops needed to apply the range of coefficients to build the space. It creates STL vectors representing the

numerator (New_Numerator) and the coefficients (New_Coefficients) of the Alpha object to be built, initializing them to the values of each related member in Last_Alpha. It sets the element of the New_Coefficients vector at Layer (meaning the coefficient on the element of Common_Basis_Alphas_ which goes into the linear combination) to the value in the loop. It then adds the index of the for loop, which represents the coefficient value, multiplied by the element of Common_Basis_Alphas_ to each element of New_Numerator. Once those values have been added to New_Numerator, the method then calls vector<int> Adjust_Alpha_Range(vector<int> Alpha_Numerator) to adjust the values added to fit the the correct range for phases. Finally, a new Alpha object is created from New_Numerator and New_Coefficients, and is passed to Add_Alphas along with an incremented Layer variable, thereby invoking the recursion. If the (Layer < Common_Basis_Alphas().size()) statement evaluates to false, then all of the loops have been nested. The method sets a variable as the denominator for the mass squared calculation, then calls the bool Linearly_Independent_Alpha(const Alpha& Last_Alpha) method, passing it Last_Alpha. If that method returns false, then Linearly_Independent_Alphas_ is set to false, indicating that this model is not consistent. Next, the method determines which type of Alpha Last_Alpha is. The method assigns values to variables Mass_Left and Mass_Right by calling the methods of the same name for Last_Alpha. A series of if statements are then used to check the criterion first for Alpha_SUSY, then Alpha_Boson, and finally Alpha_Fermion. If the length squared of the left mover is equal to 8, the length squared of the right mover is equal to 0, and the first element (the ST fermion mode) is equal to zero, then it is an Alpha_SUSY, and Last_Alpha is cast as an Alpha_SUSY, then is inserted onto the mem-

ber variables Alpha_SUSYs_ as well as Alpha_Fermions_, since the SUSY-generating sector can also generate non-SUSY partner fermions. If the left mover length squared is equal to zero, and the right mover length squared is less than 16, then it is an Alpha_Boson, and Last_Alpha is cast as an Alpha_Boson type and inserted into the member Alpha_Bosons_. If the left mover length squared is less than or equal to 8, the right mover length squared is less than or equal to 16, and the first element (the ST fermion mode) is not equal to zero, then it is an Alpha_Fermion, and Last_Alpha is cast as an Alpha_Fermion, then inserted into Alpha_Fermions_. Schematically, the process is drawn out in Figure A.2.4.

vector<int> Adjust_Alpha_Range(vector<int> Alpha_Numerator)

Once the basis alphas have been added with their coefficients, the values of the elements need to be modded back into the range $\frac{N}{2} ¡ \alpha_i \leq \frac{N}{2}$. This function does that for each of the elements and returns the resulting vector to be placed as an Alpha's numerator.

bool Linearly_Independent_Alpha(const Alpha& Last_Alpha)

Determines whether or not the given Last_Alpha is the result of linearly dependent Basis_Alpha objects. This will occur if and only if the numerator is all 0's and the coefficients which produced that numerator are not all 0's. It first checks the numerator, then checks the coefficients, but this could be optimized for more efficiency at some point.

A.2.5 FF_Alpha_Fermion.hh

NAME: Alpha_Fermion

PURPOSE: Holds the data for a fermion sector. It also has a character member which indicates which subclass of Alpha is being passed to State_Builder.

OBJECTS CREATED BY: Alpha_Builder

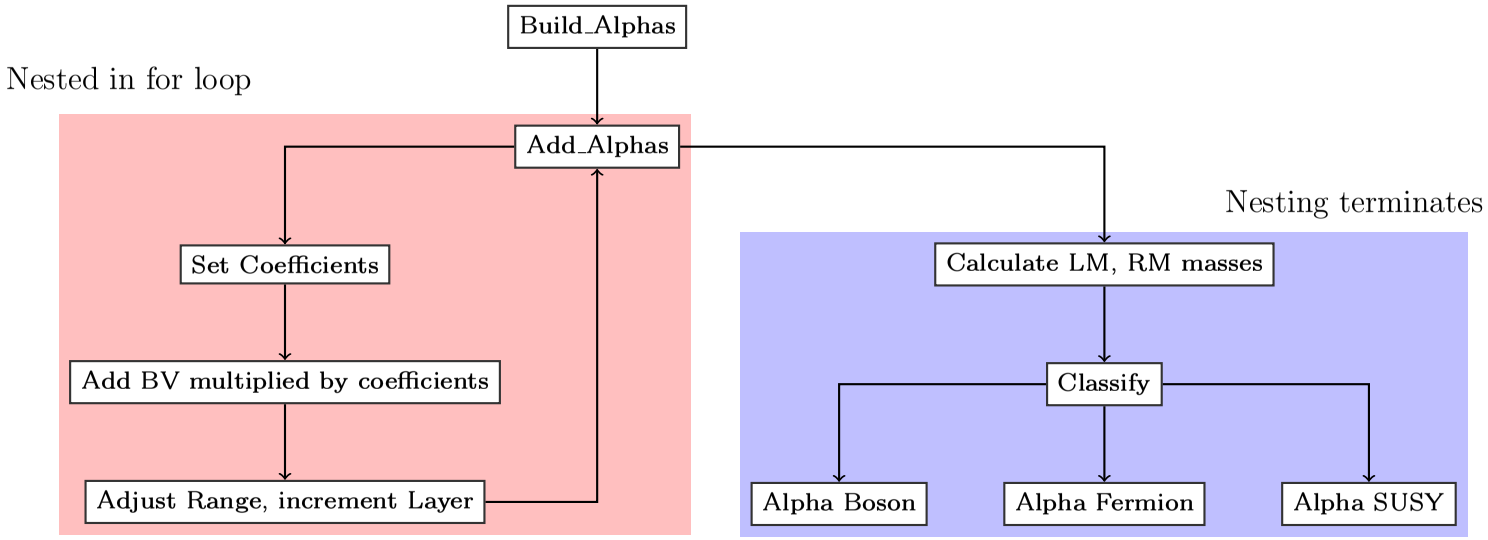


Figure A.1: The recursion tree for the **Alpha_Builder** class.

USED IN: Model_Builder, State_Builder

MEMBERS: None.

CONSTRUCTORS:

Alpha_Fermion(const vector<int>& Numerator, int Denominator,

const vector<int>& Coefficients) All arguments are passed to the Alpha class's constructor.

METHODS:

char Type() const Returns the character 'f'. Used to determine which states to build in State_Builder.

A.2.6 FF_Alpha_SUSY.hh

NAME: Alpha_SUSY

PURPOSE: Holds the data for a SUSY generating sector. It also has a character member indicating which subclass of Alpha is being passed to State_Builder.

OBJECTS CREATED BY: Alpha_Builder

USED IN: Model_Builder, State_Builder

MEMBERS: None.

CONSTRUCTORS: Alpha_SUSY(const vector<int>& Numerator,

int Denominator, const vector<int>& Coefficients) All arguments are passed to the Alpha class's constructor.

METHODS:

char Type() const Returns the character 's'. Used to determine which states to build in State_Builder.

A.2.7 FF_Basis_Alpha.hh

NAME: Basis_Alpha

PURPOSE: Holds the physical phases which the fermion modes gain when parallel-transported around the loops of space-time. A group of objects of this type specify the basis set for the modes in consistent models.

OBJECTS CREATED BY: Basis_Alpha_Builder

USED IN: Basis_Alpha_Builder, Alpha_Builder, Fermion_Mode_Map_Builder, GSO_Coefficient_Matrix_Builder, GSO_Projector

MEMBERS:

vector<int> Numerator_ The numerator values.

int Denominator_ The denominator.

int LM_Size_ The number of elements in the left moving part of the Basis_Alpha. This is necessary when computing Lorentz dot products and finding sets of matching boundary conditions.

int RM_Compact_Size_ The number of elements in the right moving compact part of the Basis_Alpha. This is necessary when finding sets of matching boundary conditions.

CONSTRUCTORS:

Basis_Alpha() This is a default constructor which does not initialize any variables. Its use is not recommended, but is needed for some of the other objects in the framework to utilize this class.

Basis_Alpha(const vector<int>& Numerator, int Denominator) This constructor initializes the Numerator_ and Denominator_ members respectively, then calls the Calculate_LM_Size() and Calculate_RM_Compact_Size() methods, which initialize LM_Size_ and RM_Compact_Size_.

Basis_Alpha(const vector<int>& Numerator, int Denominator,

const Basis_Vector& BV) Initializes Numerator_ and Denominator_ with the arguments of the same name. It then initializes LM_Size_ to BV.LM_Size() and RM_Compact_Size_ to BV.RM_Compact_Size().

Basis_Alpha(const vector<int>& Numerator, int Denominator,

int Large_ST_Dimensions) Initializes Numerator_ with Numerator and Denominator_ with Denominator. Using the value of Large_ST_Dimensions, LM_Size_ and RM_Compact_Size_ are calculated with the following equations:

$$\text{LM\_Size\_} = 28 - 2 \times \text{Large\_ST\_Dimensions} \tag{A.1}$$

$$\text{RM\_Size\_} = 2 \times (10 - \text{Large\_ST\_Dimensions}) \tag{A.2}$$

METHODS:

int Lorentz_Dot(const Basis_Alpha& Basis_Alpha_2) Returns the numerator of the Lorentz dot product between *this and Basis_Alpha_2, which is the dot product of the left moving parts minus the dot product of the right moving parts. One improvement that could be made to this function is that it be made a friend rather than a member, as it behave as an operator would.

void Display() const Displays onto the screen Denominator_ first, followed by a colon, then the numerator, with two pipes separating the left moving part from the right moving part.

void Calculate_LM_Size() Uses the size of Numerator_ to determine the size of the left moving part of the Basis_Alpha via the following equation:

$$\text{LM\_Size\_} = \frac{(\text{Numerator().size()} - 24)}{2} \tag{A.3}$$

This method could be improved by giving it a return value and an argument to prevent any possible calculation errors resulting from Numerator_ not being initialized.

void Calculate_RM_Compact_Size() This method first checks if LM_Size_ has been initialized. If it has not, the function calls Calculate_LM_Size() to initialize it. Then the method must check (effectively) the number of large space-time dimensions for the model. If there are 10 large space-time dimensions there is no compactification, and RM_Compact_Modes_ is set to zero. This occurs if and only if there are 40 elements in Numerator_. If there are greater than 40 elements in Numerator_, the method determines and initializes RM_Compact_Size_ using the following equation.

$$\text{RM\_Compact\_Size\_} = \text{LM\_Size()} - 8 \tag{A.4}$$

This method could be improved by giving it a return value and passing LM_Size_ and Numerator_.size() as arguments, rather than calling them from *this.

A.2.8 FF_Basis_Alpha_Builder.hh

NAME: Basis_Alpha_Builder

PURPOSE: Takes the basis vectors (in integer coded form, as Basis_Vector objects) and converts them to actual phase values (as Basis_Alpha objects). It builds them with and without a common denominator. Both are needed for WCFFHS model construction.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

vector<Basis_Vector> Basis_Vectors_ Holds the Basis_Vector objects that are to be converted into Basis_Alpha objects.

vector<Basis_Alpha> Basis_Alphas_ Holds the Basis_Alpha objects that do not have a common denominator.

vector<Basis_Alpha> Common_Basis_Alphas_ Holds the Basis_Alpha objects that have a common denominator.

CONSTRUCTORS:

Basis_Alpha_Builder(const Model& FFHS_Model) Takes a Model object and gets the basis vectors, placing them in the Basis_Vectors_ member.

Basis_Alpha_Builder(const vector<Basis_Vector>& Basis_Vectors)

Initializes the Basis_Vectors_ member to the argument.

METHODS:

void Build_Basis_Alphas() Converts the elements of Basis_Vectors_ into their respective Basis_Alpha objects, pushing them onto Basis_Alphas_. The method first computes the full order of the basis vector: if the basis vector has a nonzero left mover and an odd (right moving) order, it's true order is the least common multiple between two and the right moving order. This number is stored in the New_Order variable. Next, the phase value is computed. For the left movers, which are always either 0 or 1, the nonzero elements need only be set to the value of New_Order. For the right movers, a conversion factor is needed to adjust to the new denominator for the phase value. That is done with the following equation:

$$\text{Numerator\_Conversion} = \frac{2 \times \text{New\_Order}}{\text{RM Order}}. \tag{A.5}$$

The two on the right hand side of the equation is present to This is guaranteed to be an integer, because New_Order is either a multiple of

RM Order for nonzero left movers, or equal to RM Order. Once all of the right mover values have been multiplied by the conversion factor, they need to be placed in the range given by the equation

$$-\frac{\text{New\_Order}}{2} < \alpha_i \leq \frac{\text{New\_Order}}{2}. \tag{A.6}$$

Once all of the elements have been calculated and converted, the final result is cast into a Basis_Alpha object and pushed onto Basis_Alphas_.

void Build_Common_Basis_Alphas() Takes Basis_Alphas_ and converts them to a single common denominator. This step is needed to add them together when creating the Alphas for the model. First the value of the common denominator is calculated by computing the lowest common multiples between all of the elements of Basis_Alphas_. Then, a conversion factor is calculated for each of those elements using the following equation:

$$\text{Numerator\_Conversion} = \frac{\text{Common\_Denom}}{Denominator}, \tag{A.7}$$

where Common_Denom is the value of the common denominator for all of the Basis_Alphas, and *Denominator* is the denominator of the Basis_Alpha being converted.

void Display_Basis_Alphas() const Prints the elements of Basis_Alphas_ onto the screen for debugging purposes.

void Display_Common_Basis_Alphas() const Prints the elements of Common_Basis_Alphas_ onto the screen for debugging purposes.

A.2.9 FF_Basis_Vector.hh

NAME: Basis_Vector

PURPOSE: Holds the integer coded information for the fermion mode phases, its order, the number of left moving fermion modes, and the number of right moving fermion modes.

OBJECTS CREATED BY: User.

USED IN: Basis_Alpha, Basis_Alpha_Builder, Model, Model_Builder

MEMBERS:

vector<int> BV_ Stores the integer coded values for the fermion mode phases.

int Order_ Stores the order of the basis vector's right mover.

int LM_Size_ Holds the number of left moving fermion modes.

int RM_Compact_Size_ Holds the number of compact right moving modes.

CONSTRUCTORS:

Basis_Vector() A default constructor which does not initialize the members of this class. It's use is not recommended.

Basis_Vector(const vector<int>& BV, int Order) Initializes BV_ with BV and Order_ with Order. It calls Calculate_LM_Size(), then Calculate_RM_Compact_Size() to initialize LM_Size_ and RM_Compact_Size_, respectively.

Basis_Vector(const vector<int>& BV, int Order,

int Large_ST_Dimensions) Initializes BV_ with BV and Order_ with Order. It initializes LM_Size_ and RM_Size_ with the values of equations (A.1, A.2).

METHODS:

void Display() const Prints to screen order, followed by a colon, then the integer coded phase values for the basis vector. A double pipe is placed to separate the left moving part from the right moving part. For debugging purposes.

void Calculate_LM_Size() Calculates the number of left moving modes using equation (A.3). This method could be improved by giving the function a return value, and making it take an argument.

void Calculate_RM_Compact_Size() Calculates the number of right moving compact modes using equation (A.4). Could be improved by giving the function a return value and making it take an argument.

A.2.10 FF_Fermion_Mode_Map_Builder.hh

NAME: Fermion_Mode_Map_Builder

PURPOSE: Finds the sets of simultaneously matching boundary conditions for a set of basis alphas with a common denominator. Those boundary conditions are placed into an STL map with the first fermion mode in the complex pair as the key, and the second as the value.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

int Large_ST_Dimensions_ Holds the number of large space-time dimensions.

map<int, int> Fermion_Mode_Map_ Holds the indices of the complex fermion pairs. The index of the first mode in the pair is the key, the index of second mode in the pair is the value.

bool Consistent_Pairings_ A boolean flag which is set to **false** if the **Find_All_LR_Pairs(vector<int> LR_Chunk, const vector<Basis_Alpha>& Common_Basis_Alphas)** function finds that not all of the modes can be paired.

CONSTRUCTORS:

Fermion_Mode_Map_Builder() The default constructor initializes the Large_ST_Dimensions_ to zero, and the Consistent_Pairings_ variable to true. There are checks later to ensure that Large_ST_Dimensions_ initializes to a value greater than zero.

Fermion_Mode_Map_Builder(int Large_ST_Dimensions) Initializes the member Large_ST_Dimensions_ to the argument Large_ST_Dimensions, and Consistent_Pairings_ to true.

METHODS:

void Build_Fermion_Mode_Map(const vector<Basis_Alpha>&

Common_Basis_Alphas) Interface function which initializes the Fermion_Mode_Map_ member. It first determines whether or not Large_ST_Dimensions_ has been initialized to something other than zero. If it has not, it calls the Compute_Large_ST_Dimensions(const Basis_Alpha& Common_Basis_Alpha) to initialize the member. It passes the first element of Common_Basis_Alphas, as an argument for that function. Next, the methods Map_Complex_LM_Elements(const vector<Basis_Alpha>& Common_Basis_Alphas) and Map_Complex_RM_Elements(const vector <Basis_Alpha>& Common_Basis_Alphas) are called to initialize the complex elements of the map for the left and right moving parts. Those elements are always paired together for the basis vectors used by the FF_Framework to construct models. The method then pushes onto a vector (called LR_Coordinates) the indices of the elements corresponding to compact directions (y,w). This is accomplished by looping over the (x, y, w) triplets and pushing the indices of only the y's and w's. It does the same for the right moving compact modes ($\bar{y}$, $\bar{w}$), starting the loop at the left mover size plus 16 (the 16 being all complex

right moving modes) and ending at a point which satisfies the equation (A.2). Once LR_Coordinates has been filled, the method finishes by calling Find_All_LR_Pairs(vector<int> LR_Chunk, const vector<Basis_Alpha>& Common_Basis_Alphas),

which either pairs up all of the compact modes, or sets Consistent_Pairings_ to false.

void Display_Fermion_Mode_Map() const Prints the key then the value for each pair in Fermion_Mode_Map_ to the screen for debugging purposes.

int Compute_Large_ST_Dimensions(const Basis_Alpha&

Common_Basis_Alpha) Uses the following equation to compute the number of large space-time dimensions.

$$Large\_ST\_Dimensions = 14 - \frac{LM\_Size}{2} \tag{A.8}$$

This method could be improved by changing the parameter to be an integer representing the size of the left mover rather than passing a reference to the entire Basis_Alpha, which has more information than is needed.

void Map_Complex_LM_Elements(const vector<Basis_Alpha>&

Common_Basis_Alphas) Initializes the key-value pairs of Fermion_Mode_Map_ for the left moving modes which are always in a complex pair. This is done first with the space-time fermion modes, then with the x values of the triplets. The pairs are, by convention, taken to be the nearest modes in the vector, so the pairs would be $(x_1,\ x_2)$, $(x_3,\ x_4)$, etc. This method could be improved by either changing the parameter to an integer representing the number of left moving modes, or removing the parameter altogether and calculating LM_Size from Large_ST_Dimensions_. The entire vector of Basis_Alphas is not necessary.

void Map_Complex_RM_Elements(const vector<Basis_Alpha>&

Common_Basis_Alphas) Initializes the key-value pairs for the right moving modes that are always in adjacent complex pairs, namely the $\bar{\psi}$'s and $\bar{\phi}'s$. It uses the **LM_Size()** of the parameters passed to the function to determine the indices of the modes, then puts those key-value pairs into **Fermion_Mode_Map_**.

void Find_All_LR_Pairs(vector<int> LR_Coordinates,

const vector<Basis_Alpha>& Common_Basis_Alphas) First pairs the left movers with the function **Pair_LMs(vector<int> LR_Coordinates, const vector <Basis_Alpha>& Common_Basis_Alphas)**, assigning the return value (which is the unpaired indices) to **Unpaired_LMs**. It then pairs the right movers with the function **Pair_RMs(vector<int> LR_Coordinates, const vector <Basis_Alpha>&**

Common_Basis_Alphas), assigning the return value (again the unpaired indices) to **Unpaired_RMs**. This ensures that priority is given to same side pairings, which is convention for these models. The final step is to build the left-right paired modes, done with the

Pair_Mixed(vector<int> Unpaired_LMs, vector<int> Unpaired_RMs,

const vector<Basis_Alpha>& Common_Basis_Alphas). The return value (unpaired indices - there should be none for consistent models) is assigned to **Unpaired_Mixed**. If the size of **Unpaired_Mixed** is greater than 0, then there were boundary conditions which could not be paired, and the model is inconsistent. Though the consistency of the pairings is checked in other places in the framework, this check provides additional security in case the other methods weren't called.

vector<int> Pair_LMs(vector<int> LR_Coordinates,

const vector<Basis_Alpha>& Common_Basis_Alphas) Pairs as many left moving fermion modes as possible, loading them into Fermion_Mode_Map_. First, the indices for the left mover only (always exactly half the size of LR_Coordinates) are loaded into a two dimensional vector with a single row, then passed to the Find_Matching_BCs(vector<vector<int> > Matching_BCs, const vector<Basis_Alpha>& Common_Basis_Alphas) function. The return value is placed in Final_Matching_BCs_LM, which is then passed to the function Add_Pairs_To_Map(vector<vector<int> > Final_Pairs). That function returns the unpaired elements, which initialize Unpaired_LMs, which are then returned by this function. This function could be improved by merging it with Pair_RMs(vector<int> LR_Coordinates, const vector<Basis_Alpha>&

Common_Basis_Alphas), and changing LR_Coordinates into a reference.

vector<int> Pair_RMs(vector<int> LR_Coordinates,

const vector<Basis_Alpha>& Common_Basis_Alphas) The parameters passed to this method are an STL vector of integers holding the indices of the left and right fermion modes which may be paired, and an STL vector of Basis_Alpha's with a common denominator that hold the actual boundary conditions for the model. It pairs as many right moving compact modes together as possible, and returns any that cannot be paired. First, the indices of the right movers in LR_Chunk (always the second half) are placed in a single row two dimensional vector. That vector is passed as an argument, along with Common_Basis_Alphas to the Find_Matching_BCs(vector<vector<int> > Matching_BCs,

const vector<Basis_Alpha>& Common_Basis_Alphas). That function returns the sets of matching boundary conditions, but they have not yet been placed into complex pairs. That is done with the

Add_Pairs_To_Map(vector<vector<int> > Final_Pairs) function,

which returns the indices of any modes which could not be placed into a complex pair. Those modes are stored in Unpaired_RMs, which are returned.

vector<int> Pair_Mixed(vector<int> Unpaired_LMs,

vector<int> Unpaired_RMs, const vector<Basis_Alpha>&

Common_Basis_Alphas) The parameters passed to this method are two STL vectors of integers holding the indices of the unpaired left moving and right moving modes, as well as an STL vector of Basis_Alpha objects that contain the actual boundary conditions. This method takes the indices of the modes which could not be paired left-left and right-right, and puts them into left-right pairs. Firstly, it loads the indices of the unpaired LMs and RMs into a two dimensional vector with one row. It then passes that vector, along with Common_Basis_Alphas, to the Find_Matching_BCs(vector<vector<int> > Matching_BCs, const vector<Basis_Alpha>& Common_Basis_Alphas). The return value of that function is placed into a two dimensional STL vector named Final_Matching_BCs, which is then passed to the Add_Pairs_To_Map(vector<vector<int> > Final_Pairs) function. The return value of that function is assigned to Unpaired_Mixed, which is in turn returned by this function. Unpaired_Mixed should be empty for consistent models, but that is checked in Build_Fermion_Mode_Map (const vector<Basis_Alpha>& Common_Basis_Alphas). This function could be improved by merging it with Pair_LMs(vector<int> LR_Coordinates, const vector<Basis_Alpha>& Common_Basis_Alphas) and

Pair_RMs(vector<int> LR_Coordinates, const vector<Basis_Alpha>& Common_Basis_Alphas). It might also be worthwhile to pass the Unpaired_LMs and Unpaired_RMs parameters by const reference rather than by value.

vector<vector<int> > Find_Matching_BCs(vector<vector<int> > Matching_BCs, const vector<Basis_Alpha>& Common_Basis_Alphas)

The Matching_BCs parameter is the initial set of matching boundary conditions, prior to the boundary vectors being added to the model. Essentially, it should contain the indices of the compact left movers (except the already complex x's) and the indices of the compact right movers. Common_Basis_Alphas contains the actual phase values for the model with a common denominator. This function returns a two dimensional STL vector. Each row of that vector will be the indices of matching boundary values for all of the phases in Common_Basis_Alphas. The algorithm used is four nested loops. The first loop indexes Common_Basis_Alphas, stepping through the layers of the model. Within that loop, indexed by the variable CBA_Row, a two dimensional vector is created to hold the new matching boundary conditions. The second nested loop is indexed by the variable MBC_Row, and loops over the first index of Matching_Boundary_Conditions. The next nested loop is indexed by E_Val, and loops over the possible values for the phases of the model. This is the same for each row in Common_Basis_Alphas, since they all have a common denominator. Not all of the values indexed by E_Val can be actual phases in the model since each layer can have independent orders. However, this is the most complete, general treatment for this algorithm at the moment. Within that loop, a variable is created to load New_Matching_Boundary_Conditions. The innermost nested loop goes over the actual indices in the MBC_Row'th row

of Matching_Boundary_Conditions. If the value of the CBA_Row'th row of Common_Basis_Alphas at the position in Matching_Boundary_Conditions specified by MBC_Row, MBC_Column match E_Val, then the index is pushed onto New_Matching_Boundary_Conditions_Loader.

When the innermost loop terminates, if the size of New_Matching_Boundary_Conditions_Loader not zero, then the entire vector is pushed onto New_Matching_Boundary_Conditions. When the loops indexed by E_Val and MBC_Row terminate, the contents of Matching_BCs and New_Matching_Boundary_Conditions are swapped. This ensures that the matching fermion modes from each layer are taken into account along the way. This is also why Matching_BCs is not passed by reference. It would need to be copied at the beginning of the method for it to work, so no time is saved here by passing the reference. To summarize qualitatively what this method does: for each layer in the model, for each set of matching boundary conditions from previous layers, group the current layer indices according to the values of their phases for all possible phase values. Pass these new groups to the next layer, and repeat. The final step is to return the final set of matching boundary conditions for all layers in the model. The biggest improvement that could be made to this method would be to customize the loop indexed by E_Val to only go over the values possible for the layer specified by CBA_Row. This would speed up the algorithm by only looping over phase values that are allowed for that layer in the model.

vector<int> Add_Pairs_To_Map(vector<vector<int> > Final_Pairs)

The parameter for this method is a two dimensional vector which contains the indices of the matching sets of boundary conditions for the model. It returns a vector of any values which were not able to be

paired. It steps through the rows and columns of **Final_Pairs**, placing adjacent indices into **Fermion_Mode_Map_** as key-value pairs. If there are any unpaired indices left, then they are pushed onto **Unpaired_BCs**, which is returned. A consistent model will always return an empty vector, but the check for consistency in this class is in the

Find_All_LR_Pairs(vector<int> LR_Coordinates,

const vector<Basis_Alpha>& Common_Basis_Alphas).

A.2.11 FF_Group_Representation.hh

NAME: Group_Representation

PURPOSE: Holds the information for a group representation including the dimension, the triality (if it is a representation of $SO(8)$), and whether it is a complex representation or not.

OBJECTS CREATED BY: Gauge_Group

USED IN: Gauge_Group, Matter_State, Model_Builder

MEMBERS:

int Dimension_ The dimension of the representation.

char Triality_ The triality of an $SO(8)$ representation. The default value is ' ' for $SO(8)$ spinor reps as well as non-$SO(8)$ representations.

bool Is_Complex_ A boolean flag set to **true** if the representation could be complex, and **false** if it is not.

CONSTRUCTORS:

Group_Representation() A default constructor. Not recommended for explicit use.

Group_Representation(int Dimension, char Triality) Initializes Dimension_ to Dimension and Triality_ to Triality. Initializes Is_Complex_ to false.

Group_Representation(int Dimension, char Triality,

bool Is_Complex) Initializes Dimension_ to Dimension, Triality_ to Triality, and Is_Complex_ to Is_Complex.

METHODS:

void Display() const Outputs Dimension_ followed by Triality_ to the screen for debugging purposes.

friend bool operator<(const Group_Representation&

Group_Representation1, const Group_Representation&

Group_Representation2) Operator for comparing two Group_Representation objects. First compares the Dimension_ members of Group_Representation1 and Group_Representation2. If they are not the same value, the value of the integer < operator is returned. If they are the same, the Triality_ members are compared. If Group_Representation2.Triality() is equal to 'v' and Group_Representation1.Triality() is not, then the method returns true. Otherwise, it returns false.

friend bool operator==(const Group_Representation&

Group_Representation1, const Group_Representation&

Group_Representation2) Returns true if and only if the Dimension_ and Triality_ members of Group_Representation1 and Group_Representation2 are the same. Otherwise, it returns false.

A.2.12 FF_GSO_Coefficient_Matrix.hh

NAME: GSO_Coefficient_Matrix

PURPOSE: Holds the GSO coefficient matrix for an FFHS model.

OBJECTS CREATED BY: User, GSO_Coefficient_Matrix_Builder

USED IN: Model, Model_Builder, State_Builder,

GSO_Coefficient_Matrix_Builder.

MEMBERS:

vector<vector<int> > Numerators_ Holds the numerators for the GSO coefficient values.

vector<int> Denominators_ Holds the denominators for the GSO coefficient values. They are the orders of the basis vectors corresponding to the columns of the matrix.

CONSTRUCTORS:

GSO_Coefficient_Matrix() The default constructor does not initialize the members. Members are loaded through the interface functions

Load_GSO_Coefficient_Matrix_Row(const vector<int>& New_Row)

and Load_GSO_Coefficient_Matrix_Order(int New_Order).

GSO_Coefficient_Matrix(const vector<vector<int> >& Numerators,

const vector<int>& Denominators) Initializes Numerators_ with Numerators, and Denominators_ with Denominators.

METHODS:

void Load_GSO_Coefficient_Matrix_Row(const vector<int>& New_Row) Pushes

New_Row onto Numerators_. This does not push the order of the basis vector onto the matrix, because the order corresponds to the column, not the row.

void Load_GSO_Coefficient_Matrix_Order(int New_Order) Pushes New_Order onto Denominators_. New_Order corresponds to the column of the matrix, and the number of possible values for each column must correspond to New_Order (as the basis vector phase values would). This method could be improved by putting error checking to make sure the values in Numerators_ correspond to the proper orders. Currently this checking is implemented in the GSO_Coefficient_Matrix_Builder class.

void Display() const Prints Denominators_ onto the screen, then a line, then the values of Numerators_. Used for debugging.

A.2.13 FF_GSO_Coefficient_Matrix_Builder.hh

NAME: GSO_Coefficient_Matrix_Builder

PURPOSE: Takes the half of the GSO coefficient matrix specified by the user (the lower half by convention) in integer coded form, converts it to physical values, then builds the other half according to the modular invariance constraints. Also checks the matrix to ensure proper physical values have been built for the model. One improvement that could be made to this class would be to pass Common_Basis_Alphas_ as a parameter to Build_Complete_GSO_Matrix(). That would be less memory copying than having Common_Basis_Alphas_ as a member initialized in the constructors.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

GSO_Coefficient_Matrix Half_GSO_Matrix_ Holds the user specified GSO_Coefficient_Matrix object for the model, which should only have the lower half of the matrix present. If upper half values are present, they will be overwritten.

vector<Basis_Alpha> Common_Basis_Alphas_ Holds the phase values of the model with common denominators. Used for constructing the upper half of the GSO coefficient matrix using the modular invariance constraints.

vector<int> GSO_Coefficient_Orders_ Holds the orders of the basis vectors which make up the model. Used to find a common denominator for the physical values, as well as to check the consistency of the values through the modular invariance constraints.

GSO_Coefficient_Matrix Complete_GSO_Matrix_ Holds the complete, phase value form of the GSO coefficient matrix for the model. It will satisfy all of the modular invariance constraints, but may not have the proper values for the given orders. The member Consistent_GSO_Matrix_ indicates whether or not the values are consistent with the orders of the basis vectors which make up the model.

bool Consistent_GSO_Matrix_ A boolean flag which indicates whether or not the values of Complete_GSO_Matrix_ are consistent with the orders of the basis vectors. It is initialized to true in the constructors and it flagged in the Complete_Half_GSO_Matrix() method.

CONSTRUCTORS:

GSO_Coefficient_Matrix_Builder(const GSO_Coefficient_Matrix&

Half_GSO_Matrix, const vector<Basis_Alpha>& Common_Basis_Alphas) Initializes Half_GSO_Matrix_ to Half_GSO_Matrix, Common_Basis_Alphas_ to Common_Basis_Alphas, GSO_Coefficient_Orders_ to the denominators of

Half_GSO_Matrix, and Consistent_GSO_Matrix_ to true.

GSO_Coefficient_Matrix_Builder(const Model& FFHS_Model,

const vector<Basis_Alpha>& Common_Basis_Alphas) Uses FFHS_Model to initialize Half_GSO_Matrix and GSO_Coefficient_Orders. Initializes Common_Basis_Alphas_ to Common_Basis_Alphas, and Consistent_GSO_Matrix_ to true.

GSO_Coefficient_Matrix_Builder(const vector<int>&

Half_Numerators, const vector<int>& Half_Denominators,

const vector<Basis_Alpha>& Common_Basis_Alphas) Builds and initializes Half_GSO_Matrix_ using Half_Numerators and Half_Denominators. Common_Basis_Alphas_ is initialized to Common_Basis_Alphas, and Consistent_GSO_Matrix_ is initialized to true.

METHODS:

void Build_Complete_GSO_Matrix() An interface method which first converts the integer coded values of Half_GSO_Matrix_ into physical values with a common denominator by calling void Convert_Half_GSO_Matrix(), then builds the other half of the matrix by calling void Complete_Half_GSO_Matrix().

void Display_Half_GSO_Matrix() const A function which prints Half_GSO_Matrix_ to the screen for debugging purposes.

void Display_Common_Basis_Alphas() const A function which prints Common_Basis_Alphas_ to the screen for debugging purposes.

void Display_Complete_GSO_Matrix() const A function which prints Complete_GSO_Matrix_ to the screen for debugging purposes.

void Convert_Half_GSO_Matrix() This method converts the integer coded values of **Half_GSO_Matrix_** to physical values with a common denominator. The common denominator, stored in **CommonDenom**, is initialized to the common denominator in **Common_Basis_Alphas_**. Then, it loops over all but the last value of **GSO_Coefficient_Orders_**. Inside that loop, a new index is created for an internal loop over the column of **Half_GSO_Matrix_**. The columns are looped over because the order of the n^{th} column of the GSO coefficient matrix must match the order of the n^{th} basis vector in the model. That index, **Column_Start**, is initialized to the index of the outer loop if it is greater than zero, and 1 if it is zero. This sets the starting point of the next loop such that it is the first user specified element. Then the multiplicative factor which converts the integers to phase values is initialized with the following equation:

$$\textbf{Convert} = \frac{\textbf{CommonDenom}}{\textbf{GSO\_Coefficient\_Orders().at(a)}}, \tag{A.9}$$

where **a** is the outer loop index. It then multiplies the elements along that column with **Convert**, and calls **int Reset_GSO_Coefficient_Range(int Converted_GSO_Coefficient, int Converted_GSO_Denominator)**, passing it $2 \times N \times Convert$ and **CommonDenom**, respectively. This puts the integer codes into the range of physical values. The factor of two is done to satisfy the equation

$$k_{ij} = \frac{2 \times k_{ij,int}}{N}, \tag{A.10}$$

where $k_{ij,\ int}$ is the user specified integer code for the GSO coefficient matrix, and N is the common denominator for the entire basis vector set that specifies the model. When all of the user specified values of the GSO coefficient matrix have been converted, **Half_GSO_Matrix_** is reinitialized to the new values.

void Complete_Half_GSO_Matrix() Takes the physical phase values of the lower, user specified half of the GSO coefficient matrix and uses the modular invariance equations to generate the upper half and diagonal elements of the GSO coefficient matrix, initializing **Complete_GSO_Matrix_** with all of the values. Two **for** loops are nested, both spanning **Half_GSO_Matrix().Numerators().size()**. The number of row in **Half_GSO_Matrix_** will also equal the number of columns of **Complete_GSO_Matrix_** once the other elements are found. The elements are all pushed onto a two dimensional vector **Complete_GSO_Numerators**, which will be used to initialize **Complete_GSO_Matrix_**. A series of **if** statements are used to separate three cases:

a<b || b==0 This is either the (0,0) element, or a lower half element, specified by the user. It is added, unaltered, to **Complete_GSO_Numerator**, which is a loader for **Complete_GSO_Numerators.**

a==b This is a diagonal element (besides (0,0)). It is calculated through the method **int Compute_Diagonal_GSO_Element(const Basis_Alpha& The_Alpha, int First_GSO_Coefficient_Row).** It passes the element of **Common_Basis_Alphas_** which correspond to the element being calculated, as well as the first GSO coefficient element on the corresponding row. That element is always user specified. The result of that function has its range reset with **int Reset_GSO_Coefficient_Range(int Converted_GSO_Element, int Converted_GSO_Denominator)** and is then pushed onto **Complete_GSO_Numerator**. The consistency of the element is checked with **bool Check_GSO_Element_Consistency(int**

GSO_Element, int column) to make sure the new value meets the requirements for physical consistency.

a>b An upper half element which is calculated by calling int Compute_Off_Diagonal_GSO_Element(const Basis_Alpha& Alpha_a, const Basis_Alpha& Alpha_b, intHalf_GSO_Element) and passing the two elements of Common_Basis_Alphas_ corresponding to the row and column of the element being generated, as well as the corresponding user specified lower half element (with row, column reversed). The result's range is reset with int Reset_GSO_Coefficient_Range(int Converted_GSO_Element, int Converted_GSO_Denominator), and it is pushed onto Complete_GSO_Numerator. The consistency of the element is checked with bool Check_GSO_Element_Consistency(int GSO_Element, int column) to make sure the new value meets the requirements for physical consistency.

For each iteration of the outer loop, after the complete inner loop cycle, Complete_GSO_Numerator is pushed onto Complete_GSO_Numerators and cleared so that the next row can be converted and added. Finally, once all of the numerators have been adjusted and calculated, Complete_GSO_Matrix_ is initialized with the results.

int Compute_Diagonal_GSO_Element(const Basis_Alpha& The_Alpha, int First_GSO_Coefficient_Row) This method takes the Basis_Alpha corresponding to the diagonal element of the GSO coefficient matrix being computed, as well as the first element of that row in the GSO coefficient matrix. It then puts them through the equation

$$k_{ii} = \vec{\alpha}_i^B \cdot \vec{\alpha}_i^B - k_{i1} - s_i \pmod 2, \tag{A.11}$$

where the dot product between the $\vec{\alpha}_i^B$'s is a Lorentz dot product, and s_i is 1 if $\vec{\alpha}_i^B$ is a space-time fermion sector (first two elements equal 1) and 0 if $\vec{\alpha}_i^B$ is a space-time boson sector (first two elements equal 0). This equation is performed step by step in the code for clarity. First, the Lorentz dot product is computed, and the denominator multiplied by 8 is divided out. Modular invariance of the $\vec{\alpha}_i^B$'s guarantees this to be an integer. There is an extra factor of 2 in the denominator of this equation because it is done in the real, rather than complex, basis. Then, the space time value and the first GSO coefficient on the row are subtracted. Finally, the modulus is computed. While the equation is done *mod* 2, the actual computation is done *mod* $(2 \times Denom)$, so that every step in the equation results in an integral value. That way the program makes no rounding errors.

int Compute_Off_Diagonal_GSO_Element(const Basis_Alpha& Alpha_a,

const Basis_Alpha& Alpha_b, int Half_GSO_Coefficient) The parameters for this method are the two $\vec{\alpha}^B$'s corresponding to the row and column of the GSO coefficient being calculated, as well the coefficient's the lower diagonal counterpart (reversed row and column indices). It returns the upper diagonal GSO coefficient once it has been calculated. The element is calculated according the the equation

$$k_{ji} = \vec{\alpha}_i^B \cdot \vec{\alpha}_j^B - k_{ij} \qquad (\text{mod } 2). \tag{A.12}$$

For clarity, this calculation is done step by step in the code. The dot product is calculated and divided by 4 times the denominator. There is an extra factor of 2 in the division to convert from the complex basis in which the equation is typically presented to the real basis in which it is implemented. Then the corresponding lower half element of the GSO

coefficient matrix is subtracted. Finally, the mod $(2 \times Denominator)$ is applied. While the equation is *mod* 2, the denominator is inserted to keep the equation completely integral so that no rounding errors are made during the calculation.

int Reset_GSO_Coefficient_Range(int Converted_GSO_Coefficient, int Converted_GSO_Denominator) Takes the converted, but not range checked GSO coefficient and the denominator by which to set the range. This function returns the same phase, but placed in the range

$$-N < k_{ij} \leq N, \tag{A.13}$$

where N is the denominator.

bool Check_GSO_Element_Consistency(int GSO_Element, int column) This function takes the element to be checked, and the index of the column. It checks and returns the validity of the equation

$$N_j k_{ij} = 0 \pmod 2, \tag{A.14}$$

where N_j is the order of the column in the GSO coefficient matrix. In this instance, the order cannot be the common denominator for the GSO coefficients in physical form, because the values that they are allowed to take are restricted to the same values of their associated basis vectors. This method ensures that any calculated results still fit into the right values for the possible GSO coefficients. The proper order could be either the user specified order of the associated basis vector, or the least common multiple between 2 and the order of the associated basis vector. This is necessary because the user specifies only the order of the right mover, as the left mover is fixed at 2. Thus, if the left mover has nonzero elements, its order is $LCM(2, Order)$. If the left

mover is all zeros (for gauge models), then the right moving order is the order for the entire vector.

A.2.14 FF_GSO_Projector.hh

NAME: GSO_Projector

PURPOSE: Performs the GSO projections on states coming from a specified sector through a boolean interface function. Different sectors require different instances of GSO_Projector.

OBJECTS CREATED BY: State_Builder

USED IN: State_Builder

MEMBERS:

vector<Basis_Alpha> Common_Basis_Alphas_ Holds the physical values of the basis vectors making up the model with a common denominator.

char Alpha_Type_ Holds the character code specifying the sector from which the states to be tested were generated; 'f' for a fermion sector, 's' for a SUSY sector, and 'b' for a boson sector.

GSO_Coefficient_Matrix_ k_ij_ The GSO coefficient matrix for the model that is being built.

vector<int> Coefficients_ The coefficients which produced the sector that produced the states being tested.

map<int, int> Fermion_Mode_Map_ The mapping of the fermion modes between their complex (for same side) or real (for opposite side) pairs. This is needed for a correction to the GSOPs for the real fermion pairs.

CONSTRUCTORS:

GSO_Projector() A default constructor which does not initialize any values. Not recommended for explicit use.

GSO_Projector(const vector<Basis_Alpha>& Common_Basis_Alphas, char Alpha_Type, const GSO_Coefficient_Matrix& k_ij, const vector<int>& Coefficients, const map<int, int>& Fermion_Mode_Map) Initializes Common_Basis_Alphas_ to Common_Basis_Alphas, Alpha_Type_ to Alpha_Type, k_ij_ to k_ij, Coefficients_ to Coefficients, and Fermion_Mode_Map_ to Fermion_Mode_Map.

GSO_Projector(const vector<Basis_Alpha>& Common_Basis_Alphas, const Alpha& The_Alpha, const GSO_Coefficient_Matrix& k_ij, const map<int, int>& Fermion_Map) Initializes Common_Basis_Alphas_ to Common_Basis_Alphas, k_ij_ to k_ij, and Fermion_Mode_Map_ to Fermion_Mode_Map. Coefficients_ and Alpha_Type_ are initialized from The_Alpha, which is the sector that produced the states that are being tested by this instance of GSO_Projector.

METHODS:

bool GSOP(const State& The_State) const Performs the GSO projection on The_State and returns true if it passes and false if it doesn't. If Alpha_Type_ is a boson ('b'), then it calls GSOP_Boson(const State& The_State) const. If Alpha_Type_ is fermion ('f') or SUSY ('s'), then it calls GSOP_Fermion(const State& The_State) const, returning those results.

bool GSOP_Boson(const State& The_State) const Performs the GSO

projection equations optimized for boson states. Because the left movers are always the same for boson states, the equations can be simplified to save computing time. The simplified equation for boson states is

$$\sum_i k_{ji} a_i + \vec{\alpha}^B_{jR} \cdot \vec{Q}_{\vec{\alpha}R} = 0 \qquad (\textbf{mod } 2), \tag{A.15}$$

where $\vec{Q}_{\vec{\alpha}R}$ is the right moving part of the state produced by the sector $\vec{\alpha}$ and a_i is the coefficient which produced the sector $\vec{\alpha}$. The implementation of this equation requires careful treatment. Any left-right paired modes will contribute double to the dot product due to a redundancy present in the creation of the states. In the program, this is done with an additional correction term which repeats the dot products of those modes and adds the result to the projection. The entire process is wrapped in an **if** statement so that if any one of the j $\vec{\alpha}^B_j$ produces a failing result, the function will immediately return **false** without computing the other GSOPs. The modulus and additions are done such that all of the results are integral to reduce potential rounding errors. This method will return **true** only if the GSO projections are passed for all of the **Common_Basis_Alphas_**.

bool GSOP_Fermion(const State& The_State) Performs the GSO projections on the fermion and SUSY states. For those states, the equation for the GSOPs is

$$\vec{\alpha}^B_j \cdot \vec{Q}_{\vec{\alpha}} - \sum_i k_{ji} a_i - s_j = 0 \qquad (\textbf{mod } 2), \tag{A.16}$$

where $\vec{Q}_{\vec{\alpha}}$ is the state produced by the sector $\vec{\alpha}$, a_i are the coefficients of the **Common_Basis_Alphas_** which produced $\vec{\alpha}$, and s_j is 0 if $\vec{\alpha}^B_j$ is a space-time boson sector, and 1 if $\vec{\alpha}^B_j$ is a space-time fermion sector. The dot product is a Lorentz dot product with an additional factor for the left-right paired modes. There is a redundancy in the construction

of states with left-right paired modes, and the GSOP dot product must double count those modes. The process of computing the equation is wrapped by an **if** statement so that if any of the j $\vec{\alpha}_j^B$s doesn't allow $\vec{Q}_{\vec{\alpha}}$ to pass the GSOPs, then a **false** is immediately returned so that the others are not calculated. This method will return **true** only if all of the GSOPs are passed for the **Common_Basis_Alphas_**.

A.2.15 FF_Gauge_Group.hh

NAME: Gauge_Group

PURPOSE: Holds the information needed to completely specify the behavior of a gauge group in the model. It also has a method for computing the dimension of a matter state transforming under the gauge group.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder, Model, Gauge_Group_Identifier

MEMBERS:

list<State> Positive_Roots_ Holds the positive, nonzero roots for the gauge group.

State Weyl_Vector_ Holds the Weyl vector, which is half the sum of the nonzero positive roots. It is cast as a state to get access to the dot product operator for **State** objects, though it does not signify a physical state.

Gauge_Group_Name Name_ The name of the gauge group.

list<State> Simple_Roots_ The simple roots of the gauge group.

map<vector<int>, Group_Representation> Dynkin_Labels_ Holds the Dynkin labels as the key, and the representation as the value.

set<int> Complex_Rep_Dimensions_ Holds the dimensions of any complex representations the gauge group might have. Needed to set a sign convention for the complex representations.

CONSTRUCTORS:

Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name) Initializes Positive_Roots_ to Positive_Roots, and Gauge_Group_Name_ to Gauge_Group_Name. It then calls void Build_Weyl_Vector() to build Weyl_Vector_.

Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name, const list<State>& Simple_Roots) Initializes Positive_Roots_ to Positive_Roots, Name_ to Name, and Simple_Roots_ to Simple_Roots. It then calls Build_Weyl_Vector() to initialize Weyl_Vector_.

METHODS:

Group_Representation Compute_Rep_Dimension(const State& Weight, const map<int, int>& Fermion_Mode_Map) Computes the dimension of a representation under the gauge group, returning a Group_Representation object. It takes as parameters a State object representing a weight in the gauge group space, as well as Fermion_Mode_Map to distinguish the gauge charges from the left-right pairs. It first calls Compute_Dynkin_Labels(const State& Weight, const map<int, int>& Fermion_Mode_Map), storing the result in Weight_Dynkin_Labels. It then checks, via bool Not_Highest_Weight (const vector<int>& Weight_Dynkin_Labels), whether these Dynkin labels could produce a highest weight. If that returns false, then a value of 0 is returned as the representation dimension. The method calls the find

function for **Weight_Dynkin_Labels**, keeping the iterator. If the iterator is not at the end of **Dynkin_Labels_**, then the value of that iterator is returned. If it is, then **Apply_Weyl_Formula(const State& Weight, const map<int, int>& Fermion_Mode_Map)** is called to compute the dimension of the representation. Then, another check is made to ensure the Weyl formula produced a valid representation dimension (meaning it is a highest weight). If it is not a highest weight, the function returns 0. If the weight is a highest weight, then the method checks if the representation is complex. First, an **if** statement is used to determine whether the group can produce complex representations (A_N, D_N, E_6). It calls **Is_Barred_Rep(int Rep_Dimension)**. If that returns **true**, then the dimension is multiplied by -1. The triality of the representation is also declared. The method calls **bool Is_D4()** to check whether or not the triality needs to be computed. If **bool Is_D4()** returns **true**, then **char Compute_Triality**

(const vector<int>& Weight_Dynkin_Labels) is called. It's return value is set to the triality of the representation. After that, **Weight_Dynkin_Labels** and **Rep_Dimension** are added to **Dynkin_Labels_**. A **Group_Representation** object initialized with **Rep_Dimension** and **Triality** is then returned. Schematically, this process is shown in figure A.2.

bool Is_D4() const Calls the **bool Is_D4() const** method from **Name()**.

void Set_Ordered(bool New_Ordered) Sets the **Ordered_** member of **Name_** to the value of **New_Ordered**.

void Set_V_Ordered(bool New_V_Ordered) Sets the **V_Ordered_** member of **Name_** to the value of **New_V_Ordered**.

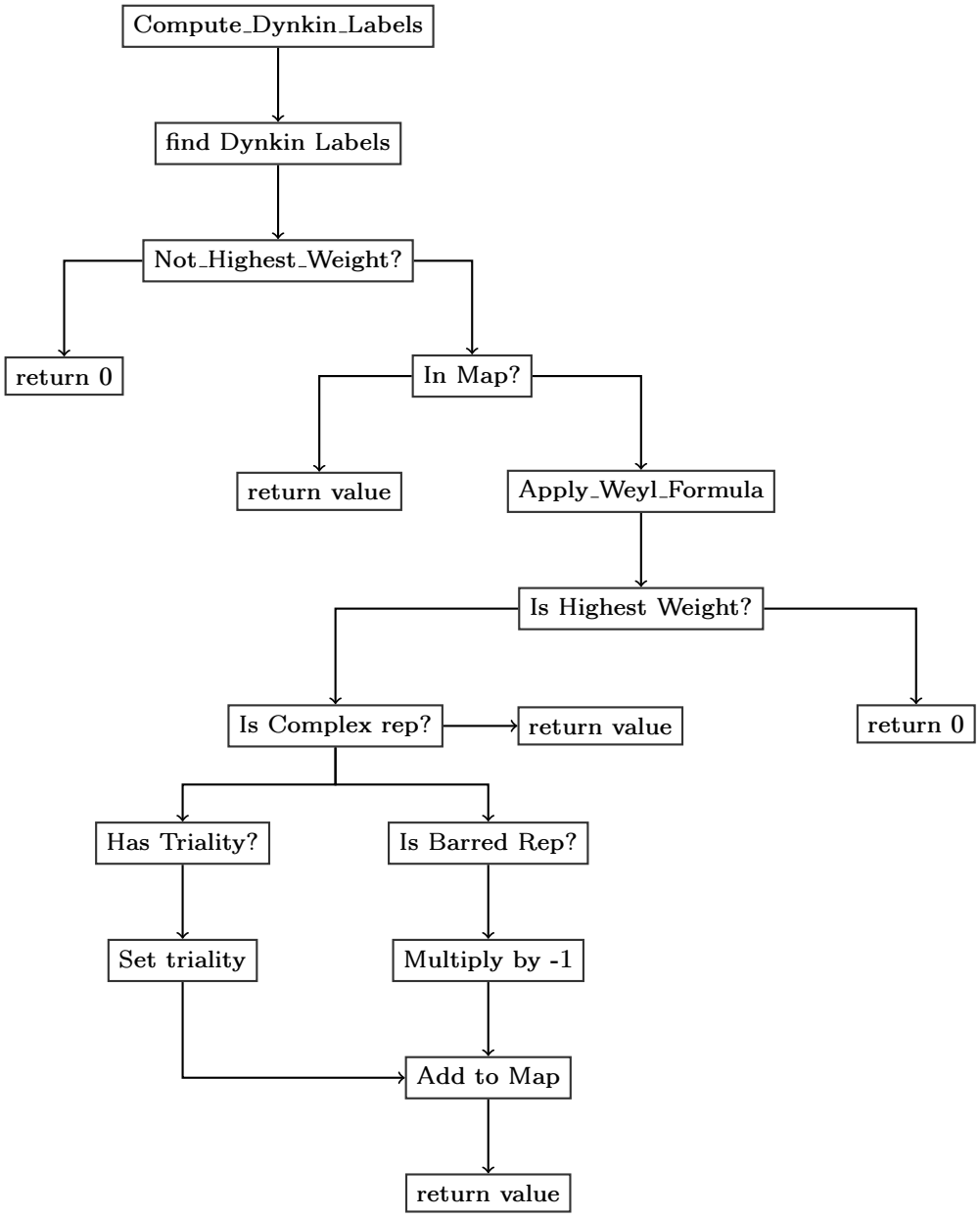


Figure A.2: A schematic of the algorithm for determining the dimension and triality (if needed) of a group representation.

void Display() const Calls the display function for **Name_**. Outputs the class, rank, and Kač-Moody level for debugging purposes.

void Display_Positive_Roots() const Displays **Positive_Roots_** on the screen by calling **Display()** for each element in the list. For debugging purposes.

friend bool operator< (const Gauge_Group& Gauge_Group1,

const Gauge_Group& Gauge_Group2) const Compares the **Name_** members of **Gauge_Group1** and **Gauge_Group2**. Used to put an ordering to the gauge groups of a model for comparison purposes.

friend bool operator==(const Gauge_Group& Gauge_Group1,

const Gauge_Group& Gauge_Group2) Compares the **Name_** members of **Gauge_Group1** and **Gauge_Group2**, returning **true** if they are equal and **false** if they are not.

void Build_Weyl_Vector() Builds the Weyl vector for the gauge group, initializing **Weyl_Vector_**. It is half the sum of the positive roots. This method is called in the constructor after **Positive_Roots_** has been initialized.

int Apply_Weyl_Formula(const State& Weight,

const map<int, int>& Fermion_Mode_Map) This method takes a reference to the state for which the representation dimension is to be determined, and the map between fermion modes. The map is needed because the dot products in the equation are only between the gauge charges; left-right pairs must be excluded from this calculation. The method applies the Weyl dimension formula for calculating the dimension of a highest weight in a representation using the nonzero positive roots of the group. The formula is

$$dim(V_{\vec{\lambda}}) = \prod_{\vec{\alpha}} \frac{\vec{\lambda}\cdot\vec{\alpha} + \vec{\rho}\cdot\vec{\alpha}}{\vec{\rho}\cdot\vec{\alpha}}, \tag{A.17}$$

where $\vec{\lambda}$ is the highest weight of the representation V, $\vec{\alpha}$ are the nonzero positive roots of the group, and $\vec{\rho}$ is the Weyl vector, equal to half the sum of the nonzero positive roots. The formula is implemented by the method by first computing the dot products, done only over the gauge charges, ignoring the left-right paired fermion modes in the state. Because the states have a common denominator, the denominators of the dot products cancel with the exception of an additional factor of 2 on $\vec{\lambda} \cdot \vec{\alpha}$ because of the $\frac{1}{2}$ on the Weyl vector's sum. The numerator and denominator of the entire equation are then multiplied by the previous iteration of the loop. Then, the method checks to see if the numerator and denominator of the formula can be reduced. If this is not done, the numbers become too large for the computer to store, and the program with throw an arithmetic or floating point exception. If the dimension can be rounded to a perfect integer, it resets the values appropriately and continues the loop. If the dimension ever becomes zero, then the entire product is zero, so the method immediately returns that value to save computing time. If the loop completes without terminating, then the dimension result is checked to ensure it is positive. The method returns 0 if the dimension is found to be negative, indicating the state is not a highest weight. If the state is a highest weight, then the dimension is returned.

int Gauge_Dot(const State& State1, const State& State2,

const map<int, int>& Fermion_Mode_Map) const Takes the two states for which the dot product is to be computed, as well as the fermion mode map to ensure only gauge charges are included in the product. The return value is the numerator of the dot product. This method performs a dot product between **State1** and **State2**'s gauge charges only.

It loops over the map starting with the right movers. Any right movers in the **first** position of **Fermion_Mode_Map** is the first element in a pair of right moving modes, which are gauge charges. Those modes in **State1** and **State2** are multiplied together, and added to the total dot product. Only the first modes in the gauge pairs are multiplied, so there is no need to correct for using a real basis when calling this method. This method could be improved by making it a **friend** to the **State** class, rather than keeping the code in **Gauge_Group**.

vector<int> Compute_Dynkin_Labels(const State& Weight,

const map<int, int>& Fermion_Mode_Map) Takes a state whose dimension under the group is to be determined, and the fermion mode map to pick out gauge charges for dot products. It returns the Dynkin labels of the state, which can indicate the dimension of its representation. This method takes the dot products of the gauge charges between **Weight** and **Simple_Roots_**, pushing them onto a vector. If at any point a gauge dot product evaluates to a number less than zero, the vector is immediately returned, since **Weight** is not the highest weight of a representation. If the loop completes, then the vector is returned.

bool Is_Barred_Rep(vector<int>& Weight_Dynkin_Labels,

int Rep_Dimension) Takes a vector of Dynkin labels and the dimension of a representation and determines whether it is barred or unbarred. There are three categories of gauge groups which admit complex representations: $A_{N>1}$, D_N, and E_6. Each is treated differently in this method, but all three use the fact that the simple roots are ordered in a specific way when the gauge group is created by **Gauge_Group_Identifier**. See the documentation for that class to learn more about this spe-

cial ordering. For $A_{N>1}$, **Weight_Dynkin_Labels** is reversed using the STL **reverse** algorithm and assigned to a new variable, **vector<int> Reversed_Weight_Dynkin_Labels**. If **Weight_Dynkin_Labels** and **Reversed_Weight_Dynkin_Labels** are not equal, then the representation is complex, and the absolute value of **Rep_Dimension** is inserted into **Complex_Rep_Dimensions_**. If **Weight_Dynkin_Labels** is less than **Reversed_Weight_Dynkin_Labels**, the representation is considered to be barred, and the method returns **true**. Otherwise, it returns **false**. For D_N, the symmetry allowing for the complex representations is between the two "final" roots in the Dynkin diagram with the exception of $SO(8)$. This method only identifies two of the three types of complex $SO(8)$ representations. The third type (defined to be the vector rep) is identified in the **char Compute_Triality(const vector<int>& Weight_Dynkin_Labels)** method separately. A new vector of Dynkin labels, called **Reversed_Weight_Dynkin_Labels** is created which is identical to **Weight_Dynkin_Labels** except that the second and third elements are switched. The second and third elements are the roots which are symmetric in the Dynkin diagram, as ordered in **Gauge_Group_Identifier**. If **Weight_Dynkin_Labels** is not equal to **Reversed_Weight_Dynkin_Labels**, the absolute value of **Rep_Dimension** is inserted into **Complex_Rep_Dimensions_**. If **Weight_Dynkin_Labels** is less than **Reversed_Weight_Dynkin_Labels** then the representation is considered to be barred, and the method returns **true**. Otherwise, it returns **false**. For E_6 there are four roots which can be inverted to yield complex representations. When the group is created, the simple roots are ordered such that the first two elements of **Simple_Roots_** are not the roots which can yield complex representations. **Weight_Dynkin_Labels**

is inverted here by switching the third and fourth element as well as the fifth and sixth element simultaneously, thus inverting the diagram. If **Weight_Dynkin_Labels** and **Reversed_Weight_Dynkin_Labels** are not equal, the the absolute value of **Rep_Dimension** is inserted into **Complex_Rep_Dimensions_**. If **Weight_Dynkin_Labels** is less than **Reversed_Weight_Dynkin_Labels** the method returns **true**. Otherwise, it returns **false**.

char Compute_Triality(const vector<int>& Weight_Dynkin_Labels)

Since the spinor reps have already been checked in **bool Is_Barred_Rep(vector<int>& Weight_Dynkin_Labels, int Rep_Dimension)**, this method only identifies the complex representation that wasn't checked in that method. Recall that **bool Is_Barred_Rep (vector<int>& Weight_Dynkin_Labels, intRep_Dimension)** inverts only the second and third elements to check for a complex representation. For $SO(8)$, the fourth root must also be checked, as it can also give complex representations. Thus, this method returns **'v'** (indicating the vector rep) if the fourth element in **Weight_Dynkin_Labels** is nonzero, and **' '** if it is zero.

bool Not_Highest_Weight(const vector<int>& Weight_Dynkin_Labels) Takes the Dynkin labels of a state, and checks to see if any are negative. If there is at least one Dynkin label that is negative, then it is not a highest weight, and the method returns **false**. Otherwise, it returns **true**.

A.2.16 FF_Gauge_Group_Identifier.hh

NAME: **Gauge_Group_Identifier**

PURPOSE: Takes the nonzero positive roots for a gauge group and identifies the group's class, rank, and Kač-Moody level. It also identifies the simple roots

of the gauge group, which are used to compute a matter representation's Dynkin labels, needed to identify the dimension of the representation.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

list<State> Positive_Roots_ Holds the nonzero positive roots of the gauge group. Initialized in the constructor.

map<int, int> Fermion_Mode_Map_ Holds the paired fermion modes in key-value pairs. Used for gauge charge dot products. Initialized in constructor.

list<State> Simple_Roots_ Holds the simple roots for the gauge group. Initialized by void Find_Simple_Roots() or void Find_Simple_Roots(char Gauge_Group_Class, int Rank).

int Long_Root_Length_Num_ The numerator's length squared of a long root for the gauge group. For Kač-Moody level 1 algebras, this is 2. Higher levels reduce this number by a factor of $\frac{1}{2}$. Initialized in void Identify_KM_Level().

CONSTRUCTORS:

Gauge_Group_Identifier(const list<State>& Positive_Roots, const map<int, int>& Fermion_Mode_Map) Initializes Positive_Roots_ to Positive_Roots and Fermion_Mode_Map_ to Fermion_Mode_Map. Sets Long_Root_Length_Num_ to 0. It will be reinitialized when void Identify_KM_Level() is called.

METHODS:

Gauge_Group Get_Group() Uses the members to identify, build, then return a **Gauge_Group** object. The method first calls **Identify_KM_Level()**, which initializes the member **Long_Root_Length_Num_**. It then calls **Identify_Class()** to identify the class of the gauge group. After that, a **switch** statement is called for the class of the group, A-G. Each case calls the appropriate **Gauge_Group Build_*_Class_Gauge_Group()** method, where * is the class.

void Display_Positive_Roots() const Displays **Positive_Roots_** onto the screen by calling the **void Display() const** function of the **State** class for each root. For debugging purposes.

void Display_Simple_Roots() const Displays **Simple_Roots_** onto the screen by calling the **void Display() const** function of the **State** class for each simple root. For debugging purposes.

void Identify_KM_Level() Identifies the Kač-Moody level of the gauge group by finding the longest length (squared) of the roots in **Positive_Roots_**. It steps through **Positive_Roots_**, looking at the length squared, and comparing it to the previous maximum length squared in the sets. The highest length squared is stored. If it is ever twice the value of the denominator, then the group is a Kač-Moody level 1 group, and **Long_Root_Length_Num_** is immediately initialized and the loop is broken since the length squared cannot be higher than 2. Such a state is massive at the string scale, and will not affect the low energy phenomenology. If the loop completes its scan of **Positive_Roots_**, then **Long_Root_Length_Num_** is initialized to the maximum value.

char Identify_Class() Identifies and returns the class of the gauge group. It loops over **Positive_Roots_**, checking the length squared of each root

against the value of **Long_Root_Length_Num_**. If any root is shorter than **Long_Root_Length_Num_**, then the gauge group is non-simply laced, and **char Identify_BCGF()** is called and returned. If all of the roots are the same length, the gauge group is simply laced, and **char Identify_ADE()** is called and returned. The processes of identification for simply and non-simply laced groups are different enough that they warrant their own functions for identification.

char Identify_ADE() Identifies a simply laced gauge group. First, it checks if the number of nonzero positive roots is degenerate between any of the groups. If the number of nonzero positive roots is degenerate, then **char Identify_ADE_Degeneracies()** identifies the group, and that result is returned. If the number of nonzero positive roots is nondegenerate, then it sets the class to **N** and checks the rank by calling **double A_Class_Rank()**. If the integer cast is equal to the double precision result, then it is an integral result, and the class is **A**. Otherwise, the class must be **D**, since all of the exceptional classes are checked in **char Identify_ADE_Degeneracies()**.

char Identify_BCGF() Identifies a non-simply laced gauge group. First, it checks if the number of nonzero positive roots is degenerate by calling **char Identify_BCGF_Degeneracies()**, which will return **N** if nondegenerate, and the class if the group is degenerate. If the group's nonzero positive roots are nondegenerate, then it counts the short roots by calling **int Count_Short_Roots()**. The number of long roots is the number of nonzero positive roots minus the number of short roots, since any group can only have two distinct root lengths. If there are more long roots, then it is a B class group. If there are more short roots, it is a C class group. If the number of short and long roots is the same, it is B_2, which is isomorphic to C_2. The FF Framework defaults to B_2 in this case.

char Identify_ADE_Degeneracies() Identifies the gauge group if the number of nonzero positive roots corresponds to two different possible groups. This is done with a **switch** statement on the number of nonzero positive roots. If there are 36 nonzero positive roots, the gauge group could be either E_6 or A_8. The method calls **void Find_Simple_Roots()**, which initializes the member **Simple_Roots**. If there are 6 simple roots, the group is E_6, and the method returns E. Otherwise, it is A_8, and the method returns A. If there are 63 nonzero positive roots, the group is E_7, and the method returns E. If there are 120 nonzero positive roots, the group could be E_8 or A_{15}. In that case, the method calls **void Find_Simple_Roots()**. If there are 8 simple roots, the group is E_8 and the method returns E, otherwise the group is A_{15} and the method returns A. If there are 210 nonzero positive roots, then the group could be either A_{20} or D_{15}. The method calls **void Find_Simple_Roots()** to determine the group's rank. If there are 15 simple roots, the group is D_{15}, and the method returns D. Otherwise, the group is A_{20}, and the method returns A. If the number of nonzero positive roots are not equal to 36, 63, 120, or 210, then it is nondegenerate, and the method returns N.

char Identify_BCGF_Degeneracies() This method is somewhat misnamed, because it only picks out the exceptional non-simply laced gauge groups. If the group is F_4 or G_2, then this method will identify it. Otherwise, **char Identify_BCGF()** will count the number of short roots and identify the group's class that way. The number of nonzero positive roots is placed in a **switch** statement. If there are 6 nonzero positive roots, then the group is G_2, and the method returns G. If there are 24 nonzero positive roots, the group is F_4, and the method returns F. If the number of nonzero positive roots is neither 6 nor 24, then the method returns

N. This method could be improved by renaming it - the method does not actually resolve degeneracies.

void Find_Simple_Roots() Finds the simple roots of a gauge group given the nonzero positive roots. This method finds the roots if the class of group is unknown, which only occurs for simply-laced gauge groups. Non simply-laced gauge groups need a more careful treatment when finding the simple roots because the number of short roots affects the validity of the simple roots that are chosen. That is done in **void Find_Simple_Roots(char Gauge_Group_Class, int Rank)**. This method finds the simple roots by building the largest possible list of mutually zero/negative dot products, making sure they are "connected" -i.e. do not form orthogonal subsets. To do this, the **sort** function is first called for **Positive_Roots_**. The order in which the nonzero positive roots appear will affect the results of the algorithm, and ordering them according the the < operator for the **State** class ensures the proper results will be found. Then, for each **State** in **Positive_Roots_**, a list is built by taking the dot product (gauge charges only, no left-right pairs) of that state with all other states in **Positive_Roots_**, keeping only those that are negative or zero. Then that list is looped over, starting with the next state in the list, again taking the dot products over the gauge charges and erasing any that are positive. That loop is repeated until the only states left have negative or zero dot products between them. If the states are connected (checked with **bool Connected_Simple_Roots(const list<State>& Simple_Roots)**) and there are more states than currently stored in **Simple_Roots_**, then that list will replace **Simple_Roots_**. This is repeated until every state in **Positive_Roots_** is used to build the initial list. The process is summarized in Figure A.3.

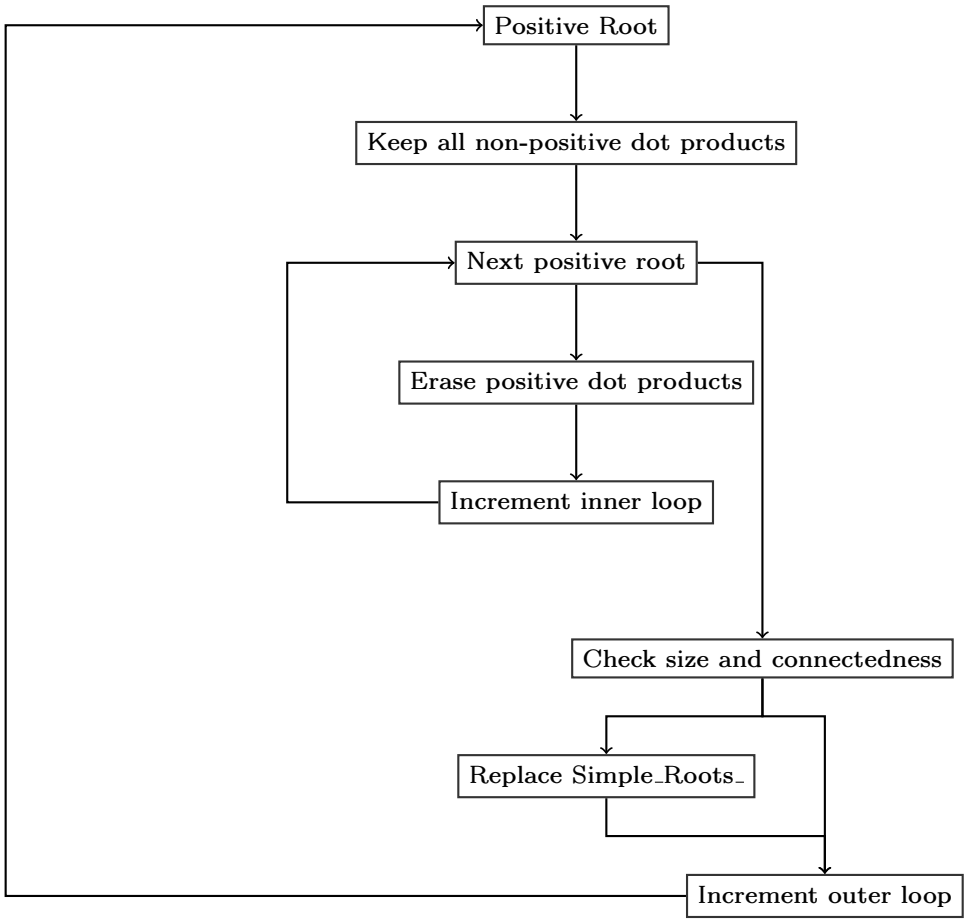


Figure A.3: A schematic of the algorithm used to find the simple roots of a gauge group.

void Find_Simple_Roots(char Gauge_Group_Class, int Rank) Finds the simple roots of a gauge group when the class and rank are already known, as for non-simply laced gauge groups. The number of short and long simple roots matters for these groups, and this method ensures that the proper long and short roots get placed in **Simple_Roots_**. It takes as parameters the gauge groups class and rank, as both are needed to determine how many short and long roots belong in **Simple_Roots_**. It should be noted that this method will work with simply laced gauge groups as well, but the extra checks for the short roots make this method less efficient than **void Find_Simple_Roots()**. The method begins first by sorting **Positive_Roots_**, then determining the number of short roots, stored in the variable **Short_Root_Count**. For B_N and G_2, the number of short roots in **Simple_Roots_** should be 1. For F_4, the number of short roots in **Simple_Roots_** should be 2. For C_N, the number of short roots in **Simple_Roots_** should be one less than the rank of the group. For a simply laced group, there are no short roots in **Simple_Roots_**. The **sort()** function is called because ordering **Positive_Roots_** according to the $<$ operator for the **State** class ensures the algorithm will work properly, as the order of **Positive_Roots_** can affect the outcome of the algorithm. The method functions in a similar way to **void Find_Simple_Roots()**, but it also keeps track of the number of short roots which have been added to the list. For each **State** in **Positive_Roots_**, a list is built of all states having a negative or zero gauge charge dot product with the specified state. If the state used to first build the list is a short root, then **Short_Roots_Added** is incremented. Additionally, any short root with a negative or zero gauge charge dot product also increments **Short_Roots_Added**. If **Short_Roots_Added** ever equals or

exceeds **Short_Root_Count**, then it is not added to this list. Once the first list is built, it is looped over. Starting with the next state in the list, take dot products of the gauge charges and erase any positive dot products. That loop is repeated until all of the states have either negative or zero dot products. When that is completed, three things are checked before swapping the contents of **Simple_Roots_** with this new list. Firstly, **bool Connected_Simple_Roots(const list<State>& Simple_Roots)** is called to make sure there weren't any mutually orthogonal subsets of simple roots found. Then, the number of short roots added must equal the number of short roots that are supposed to be in **Simple_Roots_**. Finally, it must be larger than **Simple_Roots_**. If all of these conditions are met, then **Simple_Roots_** is swapped with the new list. This process is repeated with every root in **Positive_Roots_** as the starting point for the list. This process is schematically identical to **void Find_Simple_Roots()**, which is shown in figure A.3. The additional checks for short roots are performed in the outer loop and only affect the initial list of non-positive dot products.

bool Connected_Simple_Roots(const list<State>& Simple_Roots)

Takes the list of potential simple roots and makes sure they are "connected" - that is, it checks that the simple roots are not forming mutually orthogonal subsets of simple roots. This is important because Dynkin diagrams of all Lie algebras show that no irreducible group has mutually orthogonal subsets of simple roots. To determine this, the method first places all of the potential simple roots into a list called **Unconnected_Simple_Roots**. It adds one root to another list, called **Connected_Simple_Roots**, erasing that root from **Unconnected_Simple_Roots**. The method then loops over **Connected_Simple_Roots**, and within that

loops over **Unconnected_Simple_Roots**. Inside the inner most loop, if the current element of **Connected_Simple_Roots** has a nonzero dot product with an element of **Unconnected_Simple_Roots**, then that root (from **Unconnected_Simple_Roots**) is erased from that container, and added to **Connected_Simple_Roots**. Both loops are done via C++ STL **iterator** s, which allows the containers to change dynamically without invalidating the loop limits. Each increment over **Connected_Simple_Roots** must find a connection, or the outer loop terminates. To make sure they are all connected, the method checks the size of **Connected_Simple_Roots** against the size of the argument **Simple_Roots**. If they match, it returns **true**. If they do not match, the method returns **false**.

void Order_Simple_Roots(char Class) Imposes an ordering on **Simple_Roots_** that is used by the **Gauge_Group**'s method for finding a group representation's dimension. The general idea behind this method is that the "special" roots are picked out. In the case of an A-class group (with rank greater than 1), that is either end root. In the case of a D-class group or E_6, that root is the simple root with three nonzero dot products. The idea is to pick the first root in the simple roots as an "anchor," then find any that have the proper symmetries to produce complex representations. This ordering convention allows the gauge groups to perform the correct symmetry transformations on the Dynkin diagram to ensure the correct complex representations are identified. For an A-class group, this means simply finding the one of the end roots and building the diagram from there. An outer **for** loop is initialized over **Simple_Roots_**. A loop over **Simple_Roots_** inside of the outer loop is used to count the number of nonzero dot products the simple root in the outer loop has with the others. If that number evaluates to 1, the root in question is

erase from Simple_Roots_ and the outer loop is broken. The end simple root is stored in a variable called First_Simple_Root, which will be used to order the others. To find First_Simple_Root for D-class groups and E_6, the same process is used with one exception; First_Simple_Root is now initialized when there are 3 nonzero dot products with the other simple roots. This puts a priority on the "junction" root in the D-class and E_6 Dynkin diagrams when ordering the simple roots. The roots which have a nonzero dot product with First_Simple_Root are also stored in a vector called Special_Simple_Roots. When the loop for initializing First_Simple_Root has terminated, First_Simple_Root is pushed onto a new list, Ordered_Simple_Roots. If the method has also initialized Special_Simple_Roots (for D-class and E_6 groups), then Simple_Roots_ is looped over again, this time testing the dot products of the elements in Special_Simple_Roots with the remaining Simple_Roots_ (recall that the "anchor" root has been erased from Simple_Roots_). Two of the roots will be orthogonal to the other remaining roots if the group is D-class, and one of the roots will be orthogonal to the remaining simple roots if the group is E_6. Those special roots which are orthogonal to the other simple roots are pushed onto Ordered_Simple_Roots and erased from Simple_Roots_. Finally, Ordered_Simple_Roots is looped over, and any elements of Simple_Roots_ which have a nonzero dot product with the root indexed in Ordered_Simple_Roots is erased from Simple_Roots_ and pused onto Ordered_Simple_Roots. The final step in the method is swapping Simple_Roots_ with Ordered_Simple_Roots.

int Gauge_Dot(const State& State1, const State& State2) This method computes the dot products of the gauge charges of State1 and State2. It returns the numerator of that dot product only. It picks out the gauge charges

using Fermion_Mode_Map_. A right moving mode that is the first member of a complex fermion pair (which is never true for left-right pairings) is a gauge charge. This method only performs the dot product for the first element in the fermion mode pair, so no corrections are needed for the dot products due to the use of a real basis.

double A_Class_Rank() This method returns the projected rank of an A class gauge group using the size of Positive_Roots_. If the return value can be cast as an integer and not change, then the group is an A class (excluding redundancies with E and D class gauge groups) group. Otherwise it is not. The formula for the rank of an A class gauge group given the number of nonzero positive roots is given below:

$$Rank = \frac{(\sqrt{1 + 8 \times \text{Positive\_Roots().size()}} - 1)}{2}. \tag{A.18}$$

int Count_Short_Roots() Counts and returns the number of roots in Positive_Roots_ which have length squared less than Long_Root_Length_Num_.

Gauge_Group Build_A_Class_Group() Builds and returns an A class gauge group once Positive_Roots_ and Simple_Roots_ have been initialized. First, the Kač-Moody level must be calculated. This is done according to the equation

$$KM\ Level = \frac{2}{L}, \tag{A.19}$$

where L is the length squared of the longest root in the gauge group. Since the length-squared of the numerator is stored in Long_Root_Length_Num_, the equation is calculated using

$$\text{KM\_Level} = \frac{2 \times \text{Positive\_Roots().front().Denominator()}^2}{\text{Long\_Root\_Length\_Num\_}}. \tag{A.20}$$

Since the denominators of all the states in Positive_Roots_ are the same, the front of the list is used for its denominator. Once the Kač-Moody

level has been calculated, the method checks to see if **Simple_Roots_** has been initialized. If they were needed to resolve a redundancy in identifying the class of the group, then this will evaluate to **true**, and a **Gauge_Group_Name** object will be created with the constructor **Gauge_Group_Name(char Class, int Rank, int KM_Level)**, the rank being the size of **Simple_Roots**. If the rank is greater than 1, the simple roots must be ordered to impose a convention for any complex matter representations in the model. This is done by calling **void Order_Simple_Roots(char Class)**, passing it 'A'. This **Gauge_Group_Name** is passed to the **Gauge_Group** constructor **Gauge_Group(list<State> Positive_Roots, Gauge_Group_Name Name, list<State> Simple_Roots)** along with **Positive_Roots_** and **Simple_Roots_**. If **Simple_Roots_** has not been initialized, then the method first calculates the rank of the group by calling **double A_Class_Rank()**. It then calls **void Find_Simple_Roots(char Class, int Rank)**, passing it the class and the rank. If the rank is greater than 1, the simple roots are ordered via **void Order_Simple_Roots(char Class)**. Then, as above, the name is created and passed to the constructor for **Gauge_Group**. The **Gauge_Group** object is then returned.

Gauge_Group Build_B_Class_Group() Uses **Positive_Roots_** and **Simple_Roots_** to evaluate the group's rank and Kač-Moody level. This is done according to equations (A.19, A.20). Once that has been completed, the method checks the size of **Simple_Roots_** to see if it has been initialized. It should not be, but this is a safety check in case the code is altered in some way. If **Simple_Roots_** has not been initialized, then the method calls **void Find_Simple_Roots(char Class, int Rank)**. A **Gauge_Group_Name** object is then created by passing the constructor **Gauge_Group_Name(char Class, int Rank, int KM_Level)** the class, the square root of the size of **Positive_Roots_**

which is the formula for the rank, and the calculated Kač-Moody level. The **Gauge_Group_Name** object is then passed to the constructor **Gauge_Group(list<State> Positive_Roots, Gauge_Group_Name Name, list<State> Simple_Roots)**. That **Gauge_Group** object is returned.

Gauge_Group Build_C_Class_Gauge_Group() Uses **Positive_Roots_** and **Simple_Roots_** to compute the Kač-Moody level and rank of a gauge group, returning a **Gauge_Group** object of Cartan class C_N. First, the Kač-Moody level is calculated using equations (A.19, A.20). The rank is also calculated, and is equal to the square root of the number of positive roots. A **Gauge_Group_Name** object is then created using the constructor **Gauge_Group_Name(char Class, int Rank, int KM_Level)**, passing it '**C**', the calculated rank, and the calculated Kač-Moody level, respectively. The method the checks to see if **Simple_Roots_** has been initialized. In the current form, the program should not initialize it, but if the code gets changed this check makes sure additional resources are not used to find something which has already been initialized. If **Simple_Roots_** has not been initialized, then the method calls **void Find_Simple_Roots(char Class, int Rank)**, passing it '**C**' and the rank. When that is completed the method creates a **Gauge_Group** object using the constructor **Gauge_Group(list<State> Positive_Roots, Gauge_Group_Name Name, list<State> Simple_Roots)**, passing it **Positive_Roots_**, the **Gauge_Group_Name** object created earlier, and **Simple_Roots_**. The **Gauge_Group** object is then returned.

Gauge_Group Build_D_Class_Group() Uses **Positive_Roots_** and **Simple_Roots_** to calculate the rank and Kač-Moody level of a D class gauge group. It creates said group and returns it. First, the method computes the Kač-Moody level of the group using equations (A.19, A.20). It then

computes the rank from the number of positive roots using the formula

$$Rank = \frac{1 + \sqrt{1 + 4 \times NZPR}}{2}, \tag{A.21}$$

where NZPR are the number of nonzero positive roots. Once the rank and Kač-Moody level have been determined, the method creates a **Gauge_Group_Name** object using the **Gauge_Group_Name(char Class, int Rank, int KM_Level)**, passing it 'D', the rank, and the Kač-Moody level. After the **Gauge_Group_Name** has been created, the method then checks to see if **Simple_Roots** has been initialized. If it hasn't, then **void Find_Simple_Roots(char Class, int Rank)** is called with 'D' and the rank passed as parameters, respectively. The simple roots are then ordered using **void Order_Simple_Roots(char Class)**. A **Gauge_Group** object is then created using the constructor **Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name, const list<State>& Simple_Roots)**, passing it **Positive_Roots_**, the **Gauge_Group_Name** object created earlier, and **Simple_Roots_**.

void Build_E_Class_Group() Uses **Positive_Roots_** and **Simple_Roots_** to find an E class gauge group's rank and Kač-Moody level, then builds and returns a **Gauge_Group** object. The method starts by calculating the Kač-Moody level of the group using equations (A.19, A.20). Since the group is exceptional, it can be only one of three possible ranks: 6,7, or 8. E_6 and E_8 are degenerate to other simply-laced groups (have the same number of nonzero positive roots), so their simple roots were already found to resolve the degeneracy. The method uses a **switch** statement on the size of **Simple_Roots_** to determine the rank. The default value is 7. If the **switch** evaluates to the **default** case, the method checks to see if **Simple_Roots_** has been initialized. If that is not true, then **void**

Find_Simple_Roots(char Class, int Rank) is called, passing it E and 7, respectively. If the **switch** evaluates to 8, then the rank is set accordingly. If the **switch** evaluates to 6, the rank is set accordingly and **void Order_Simple_Roots(char Class)** is called to impose an ordering for any complex matter representations the group will have. Once the rank has been established, a **Gauge_Group_Name** object is created using the constructor **Gauge_Group_Name(char Class, int Rank, int KM_Level)**, with E, the rank, and the Kač-Moody level of the group. After that, a **Gauge_Group** object is created using the constructor **Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name, const list<State>& Simple_Roots)**, passing **Positive_Roots_**, the **Gauge_Group_Name** created earlier, and **Simple_Roots_**. The **Gauge_Group** is returned.

void Build_F_Class_Group() Uses **Positive_Roots_** and **Simple_Roots_** to determine the Kač-Moody level of the group F_4, which is built and returned. The Kač-Moody level is determined using equations (A.19, A.20). Since the only F class group is F_4, then a **Gauge_Group_Name** object is created using the constructor **Gauge_Group_Name(char Class, int Rank, int KM_Level)**, passing it F, 4, and the calculated Kač-Moody level. After that, if **Simple_Roots_** has not been initialized, the method calls **void Find_Simple_Roots(char Class, int Rank)**, passing it F and 4, respectively. Finally, the method creates the **Gauge_Group** object using the constructor **Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name, const list<State>& Simple_Roots)**, passing **Positive_Roots_**, the **Gauge_Group_Name** object created earlier, and **Simple_Roots_**. The **Gauge_Group** object is returned.

void Build_G_Class_Group() This method uses **Positive_Roots_** and **Simple_Roots_** to determine the Kač-Moody level of the group G_2, which is built and

returned. The Kač-Moody level is determined using equations (A.19, A.20). Since there is only one G class group, G_2, a **Gauge_Group_Name** object is created using the constructor **Gauge_Group_Name (char Class, int Rank, int KM_Level)**, passing it 'G', 2, and the calculated Kač-Moody level. After that, if **Simple_Roots_** has not been initialized, the method calls **void Find_Simple_Roots(char Class, int Rank)**, passing it 'G' and 2. Then a **Gauge_Group** object is created using the constructor **Gauge_Group(const list<State>& Positive_Roots, Gauge_Group_Name Name, const list<State>& Simple_Roots)**, passing **Positive_Roots_**, the previously created **Gauge_Group_Name** object, and **Simple_Roots_**. The **Gauge_Group** object is then returned.

A.2.17 FF_Gauge_Group_Name.hh

NAME: Gauge_Group_Name

PURPOSE: Holds the data needed to specify a gauge group's name - the class (in Cartan notation), the rank, and the Kač-Moody level. Also holds information for catgorizing gauge groups for model comparison.

OBJECTS CREATED BY: Gauge_Group_Identifier

USED IN: Gauge_Group_Identifier, Gauge_Group

MEMBERS:

char Class_ Holds the Cartan classification for the group name.

int Rank_ Holds the rank for the group name.

int KM_Level_ Holds the Kač-Moody level for the group name.

bool Ordered_ A boolean flag indicating that the complex representations have been ordered.

bool V_Ordered_ A boolean flag indicating that the vector representations of $SO(8)$ have been ordered.

CONSTRUCTORS:

Gauge_Group_Name() Default constructor for this class. Sets **Class_** to 'N', **Rank_** to 0, and **KM_Level_** to 0. **Ordered_** and **V_Ordered_** are initialized to **false**. Explicit use of this constructor is not recommended.

Gauge_Group_Name(char Class, int Rank, int KM_Level) Initializes **Class_** to **Class**, **Rank_** to **Rank**, and **KM_Level_** to **KM_Level**. **Ordered_** and **V_Ordered_** are initialized to **false**.

METHODS:

bool Is_D4() const Returns **true** if the group is D_4, **false** otherwise.

void Display() const Displays the class, the rank, and the Kač-Moody level of the group name onto the screen for debugging purposes.

bool operator<(const Gauge_Group_Name Gauge_Group_Name2) const Compares ***this** to Gauge_Group_Name2 by first comparing the class, then the rank, then the Kač-Moody level. This method could be improved by making it a **friend** function rather than a member.

bool operator==(const Gauge_Group_Name Gauge_Group_Name2) const Returns **true** if and only if the class, rank, and Kač-Moody level of ***this** match those of **Gauge_Group_Name2**. This method could be improved by making it a **friend** function rather than a member.

A.2.18 FF_Math.hh

NAMESPACE: FF

PURPOSE: This file defines a namespace for basic mathematical functions needed within the framework.

USED IN: Basis_Alpha_Builder, GSO_Coefficient_Matrix_Builder, Model, Modular_Invariance_Checker.

METHODS:

int GCD(int a, int b) Returns the greatest common divisor between a and b. Uses Euclid's algorithm.

int LCM(int a, int b) Returns the least common multiple of a and b according to the formula

$$LCM = \frac{a \times b}{GCD(a,b)} \tag{A.22}$$

Where $GCD(a,b)$ is the greatest common divisor between a and b.

A.2.19 FF_Matter_State.hh

NAME: Matter_State

PURPOSE: Inherits from the State class, adding a member for holding the dimensions of the group representations under which the state transforms.

OBJECTS CREATED BY: Model_Builder

USED IN: Model, Model_Builder

MEMBERS:

vector<Group_Representation> Representations_ Holds the group representations under which the state transforms.

CONSTRUCTORS:

Matter_State() Default constructor which does not initialize any members. Not recommended for explicit use.

Matter_State(const vector<int>& Numerator, int Denominator, int LM_Size,

vector<Group_Representation>& Representations) Takes the numerator, denominator, the size of the left mover, and the group representations under which the state transforms. Initializes Representations_ to Representations, and passes Numerator, Denominator, and LM_Size to the base class constructor State(const vector<int>& Numerator, int Denominator, int LM_Size).

METHODS:

void Display_Representations() const Prints the contents of Representations_ to the screen for debugging purposes.

bool operator<(const Matter_State& Matter_State2) const Compares the Representations_ member of *this to Matter_State2. This method could be improved by making it a friend function rather than a member function.

bool operator==(const Matter_State& Matter_State2) const Returns true if and only if the Representations_ member of *this is identical to the Representations_ member of Matter_State2. This method could be improved by making it a friend function rather than a member function.

A.2.20 FF_Model.hh

NAME: Model

PURPOSE: Holds the information needed to completely specify a free fermionic heterotic string model. Also provides methods for loading the model with inputs.

OBJECTS CREATED BY: Model_Builder

USED IN: Basis_Alpha_Builder, Fermion_Mode_Map_Builder, GSO_Coefficient_Matrix_Builder, Model_Builder

MEMBERS:

vector<Basis_Vector> BV_Set_ The integer coded basis vectors for the model.

GSO_Coefficient_Matrix k_ij_ The GSO coefficient matrix for the model.

vector<Gauge_Group> Gauge_Groups_ The gauge groups for the model.

list<Matter_State> Matter_States_ The matter states for a model.

list<State> SUSY_States_ The gravitino states in a model, indicating the number of ST SUSYs for models in all but 8 large ST dimensions.

int U1_Factors_ The number of $U(1)$ factors in a model.

bool NAHE_Loaded_ A boolean flag indicating that the NAHE set or NAHE variation have been loaded into the model. This flag activates certain optimizations in the other classes for systematic searches. Ususally not needed for individual model construction.

CONSTRUCTORS:

Model() Default constructor which does not initialize anything. Not recommended for explicit use.

Model(int Large_ST_Dimensions) Calls void Load_One_Vector(int Large_ST_Dimensions) to initialize the all periodic basis vector, then sets NAHE_Loaded_ to false. The other members are initialized either by loader functions or through Model_Builder.

METHODS:

void Load_S_Vector(int Large_ST_Dimensions) Uses the number of large space-time dimensions (D) to build the gravitino generating basis vector, usually labeled **S**. The $D-2$ complex space-time fermions are initialized first to periodic values, followed by the x values of the $10-D$ compactified triplets. The y and w values of those triplets are anti

periodic. The remaining $52 - 2D$ right movers are initialized to zero. A Basis_Vector object is created using the constructor Basis_Vector(const vector<int>& BV, int Order, int Large_ST_Dimensions), passing it the vector created above, 2 for the order, and Large_ST_Dimensions. The Basis_Vector is then pushed onto BV_Set_ using void Load_BV_Set(const Basis_Vector& New_BV).

void Load_BV_Set(const Basis_Vector& New_BV) Pushes New_BV onto BV_Set_. Also calls void Load_GSO_Coefficient_Matrix_Order(int New_Order) to add the order of New_BV to k_ij_.

void Load_k_ij_Row(const vector<int>& New_k_ij_Row) Calls void Load_GSO_Coefficient_Row(const vector<int>& New_Row) to put New_k_ij_Row into k_ij_.

void Add_Gauge_Group(const Gauge_Group& New_Gauge_Group) Pushes New_Gauge_Group onto Gauge_Groups_.

void Sort_Gauge_Groups() Sorts the elements of Gauge_Groups_ according to their Name_ members (handled by Gauge_Group and Gauge_Group_Name). This is done with the STL algorithm sort function.

void Add_Matter_State(const Matter_State& New_Matter_State) Adds New_Matter_State to Matter_States.

void Sort_Matter_States() Sorts the contents of Matter_States_ by calling the C++ list's sort() function.

void Display_BV_Set() const Displays the contents of BV_Set_ onto the screen by calling the void Display() const member function of Basis_Vector. For debugging.

void Display_k_ij() const Displays k_ij_ by calling void Display() const for the GSO_Coefficient_Matrix class. For debugging.

void Display_Gauge_Groups() const Steps through **Gauge_Groups_**, calling **void Display() const** from the **Gauge_Group** class for each element. For debugging.

void Display_Matter_Representations() const Displays the dimensions of the matter representations for the model, along with the number of copies of each set of dimensions. The actual vectors are not displayed. Since **Matter_States_** is ordered by charges, they must be placed in a form which groups the states according to their representations. The method declares a two dimensional vector holding the actual representations (**Representations**), and a one dimensional vector for holding the number of copies of each representation (**Duplicate_Representations**). **Matter_States_** is then looped over, and within that, **Representations** is looped over to see if the current state in the outer loop is already present in **Representations**. If it is, then the corresponding element of **Duplicate_Representations** is incremented. Otherwise, the **Representations_** member of the current state in the outer loop is added to **Representations**, and a 1 is pushed onto **Duplicate_Representations**. This is done only after all of the elements of **Representations** have been checked. Once all of **Matter_States_** has been counted, the results are displayed on the screen, with the number of duplicates followed by a colon, then the dimensions of the representations. For debugging.

void Display_Particle_Content() const Displays the information related to the particle content of the model onto the screen. The method first calls **void Display_Gauge_Groups() const**, followed by **void Display_Matter_Representations() const** to output the force and matter content of the model. The number of $U(1)$ factors is then displayed, followed by the number of space-time SUSYs, which corresponds to the size of **SUSY_States_**.

void Load_One_Vector(int Large_ST_Dimensions) Pushes the all periodic basis vector onto **BV_Set_**. The size of the vector is determined from the number of large space-time dimensions according to the formula

$$Size = 80 - 4 \times D \tag{A.23}$$

where D is the number of large space-time dimensions. The vector is loaded using **void Load_BV_Set(const Basis_Vector& New_BV)**.

A.2.21 FF_Model_Builder.hh

NAME: Model_Builder

PURPOSE: Uses the classes of the FF Framework to construct a free fermionic heterotic string model. Has an interface for setting the inputs, checking for modular invariance, as well as constructing and altering the model.

OBJECTS CREATED BY: User.

USED IN: Main.

MEMBERS:

Model FFHS_Model_ The **Model** object containing the data for the model itself.

bool Consistent_Fermion_Mode_Pairs_ A boolean flag which is set to **true** when all of the fermion modes can be formed into pairs, and set to **false** when they cannot. This member is initialized in the constructor and flagged in **void Build_Fermion_Mode_Map()**.

Linearly_Independent_Alphas_ A boolean flag which is set to **true** when the basis vectors are linearly independent, and set to **false** when they are not. This member is initialized in the constructor and flagged in **void Build_Alphas()**.

Consistent_GSO_Matrix_ A boolean flag which is set to **true** when the GSO coefficient matrix is consistent, and **false** when it is not. This member is initialized in the constructor and flagged in **void Build_k_ij()**.

vector<Basis_Alpha> Basis_Alphas_ The $\vec{\alpha}^B$'s for the model before the common denominator is found.

vector<Basis_Alpha> Common_Basis_Alphas_ The $\vec{\alpha}^B$'s for the model, but with a common denominator.

set<Alpha_Boson> Alpha_Bosons_ Holds the boson producing sectors for the model.

set<Alpha_Fermion> Alpha_Fermions_ Holds the fermion producing sectors for the model.

set<Alpha_SUSY> Alpha_SUSYs_ Holds the gravitino generating sectors for the model.

map<int, int> Fermion_Mode_Map_ The map between fermion mode pairs for the model.

list<State> SUSY_States_ Holds the gravitino states for the model, the number of which correspond to the number of space-time supersymmetries in all but 8 large space-time dimensions.

CONSTRUCTORS:

Model_Builder(int Large_ST_Dimensions) Takes the number of large space-time dimensions as a parameter. First creates a new **Model** object using the constructor **Model(int Large_ST_Dimensions)**, then sets **FFHS_Model_** equal to that object. Next, **Consistent_Fermion_Mode_Pairs_**, **Linearly_Independent_Alphas_**, and **Consistent_GSO_Matrix_** are initialized to **true**.

METHODS:

void Load_S_Vector(int Large_ST_Dimensions) Calls the **void Load_S_Vector(int Large_ST_Dimensions)** member function for **FFHS_Model_**, passing it **Large_ST_Dimensions.**

void Load_Basis_Vector(const Basis_Vector& New_Basis_Vector) Calls the **void Load_BV_Set(const Basis_Vector& New_Basis_Vector)** member function of **FFHS_Model_**, passing it **New_Basis_Vector.**

void Load_k_ij_Row(const vector<int>& New_k_ij_Row) Calls the **void Load_k_ij_Row(const vector<int>& New_k_ij_Row)** member function of **FFHS_Model_**, passing it **New_k_ij_Row.**

void Load_Default_k_ij() Loads a default value for the GSO coefficient matrix consisting of all periodic values. Odd ordered modes default to the nearest integer, since odd orders do not have periodic modes. The method first loads a 1 into the upper left corder for the all periodic vector, then loops over the remaining basis vectors in **FFHS_Model_**'s **BV_Set_**. A loop within that is placed to push the orders onto the row according to the column. The inner loop's size is designed to stay in the lower half of the matrix. For example, a model with an order 4, order 3, and order 2 basis vector (in that order) would have the following default k_{ij} matrix.

$$k_{ij}^{Default} = \left(\begin{array}{c|cccc} & \vec{\mathbb{1}} & O4 & O3 & O2 \\ \hline \vec{\mathbb{1}} & 1 & \square & \square & \square \\ O4 & 1 & \square & \square & \square \\ O3 & 1 & 2 & \square & \square \\ O2 & 1 & 2 & 1 & \square \end{array}\right)$$

The default k_{ij} matrix is not guaranteed to be consistent, so exercise caution when using this function. Since this method uses BV_Set_, the basis vectors need to be added before this method is called.

void Load_NAHE_Set() Loads the NAHE set into FFHS_Model_ using the NAHE_Set_Loader class, which is part of the Extended FF Framework. This method is designed to optimize systematic and individual NAHE extensions, and is not needed for general WCFFHS model construction.

void Load_NAHE_Variation() Loads the NAHE variation into FFHS_Model_ using the NAHE_Variation_Loader class, which is part of the Extended FF Framework. This method is designed to optimize systematic and individual NAHE variation extensions, and is not needed for general FFHS model construction.

bool Check_Modular_Invariance() Checks whether FFHS_Model_ has modular invariance. First, the method checks whether Common_Basis_Alphas_ has been initialized. If it has not, then it calls **void Build_Basis_Alphas()**. After that, the method creates a Modular_Invariance_Checker object, returning the value of that class's **bool Test_Modular_Invariance(const vector <Basis_Alpha>& Basis_Alphas)**, with Common_Basis_Alphas_ passed to that function as an argument.

bool Check_Linear_Independence() This method tests the inputs to ensure they are linearly independent. If the inputs are not linearly independent, then the model is inconsistent. It first checks to see if Basis_Alphas_ has been initialized, since the method needs Alphas_ to determine the linear independence. If it has not been initialized, then it calls **void Build_Basis_Alphas()** and **void Build_Alphas()**. If Basis_Alphas_ has been initialized, but Alpha_Bosons_, Alpha_Fermions_, and Alpha_SUSYs_ have not,

then the method only calls **void Build_Alphas()**. **void Build_Alphas()** sets the flag **Linearly_Independent_Alphas_**, which is returned by this method.

bool Check_k_ij_Consistency() Checks the consistency of the GSO coefficient matrix in **FFHS_Model_**. The method first determines if **Common_Basis_Alphas_** has been initialized. If it has not, the method calls **void Build_Alphas()**. Then, the method checks whether the GSO coefficient matrix in **FFHS_Model_** has been completed. This is done by comparing the first row's size with the number of rows. If they are equal, then it is a square matrix, otherwise it has not yet been completed. If the matrix hasn't been completed, the method calls **void Build_k_ij()**, which builds the GSO coefficient matrix and flags **Consistent_GSO_Matrix_**. That member is then returned.

bool Check_Model_Consistency() Checks the overall consistency of a model, ny testing modular invariance, properly paired fermion modes, and a consistent GSO coefficient matrix. It returns **true** if the model is consistent, and **false** if it is not. It first checks if the model has modular invariance by calling **bool Check_Modular_Invariance()**. If it does not, then the method returns **false**. If it does, then the method calls **bool Check_k_ij_Consistency()**. If that is false, then the method returns **false**. If **bool Check_k_ij_Consistency()** returns **true**, then **bool Check_Linear_Independence()** is called, the result of which is returned.

void Build_Gauge_Group_Model() Builds the gauge groups for a WCFFHS model. First, it checks to see if **Basis_Alphas_** has been initialized. If it has not, the method calls **void Build_Basis_Alphas()**. Then the method checks to see if **Alpha_Bosons_**, **Alpha_Fermions_**, and **Alpha_SUSYs_** have been initialized. If they have not, then **void Build_Alphas()** is called. Next, the method checks if **Fermion_Mode_Map_** has been initialized. If it has not,

then **void Build_Fermion_Mode_Map()** is called. The last preliminary check the method must do is ensure that the GSO coefficient matrix has been completed. This is done by comparing the size of the first row to the number of rows. If they are the same, then the matrix is square. If they are not, then **void Build_k_ij()** is called. Once the necessary members have been initialized, the method calls **void Build_Gauge_Groups()** to build the gauge groups, and **void Compute_U1_Factors()** to determine the number of $U(1)$ factors in the model.

void Build_Model() Builds a WCFFHS model's gauge and matter content. The gauge content is constructed first via **void Build_Gauge_Group_Model()**, then the number of space-time SUSYs is determined by **void Build_SUSY_States()**. Finally, the matter states are built using **void Build_Matter_States()**.

void Display_Gauge_Group_Roots() const Displays the nonzero positive roots of the gauge groups onto the screen. For each gauge group, **void Display() const** is called for that object to output the name of the group, then the positive roots are looped over, printing to screen. For debugging.

Model& rFFHS_Model() This is a special accessor which returns a reference (rather than a **const** reference) to **FFHS_Model_**. This allows for other methods to access and alter **FFHS_Model_**. After the method is called, the consistency of the model is not rechecked, so use caution when calling this function.

void Build_Basis_Alphas() Constructs the basis alphas from the basis vectors. It initializes **Basis_Alphas_** and **Common_Basis_Alphas_**. The method begins by creating a **Basis_Alpha_Builder** object using the constructor **Basis_Alpha_Builder(const Model& FFHS_Model)**, passing it **FFHS_Model_**. The

method then calls void Build_Basis_Alphas() and void Build_Basis_Alphas() from the Basis_Alpha_Builder. It sets Basis_Alphas_ to the Basis_Alphas_ member of the Basis_Alpha_Builder, and Common_Basis_Alphas_ to the Common_Basis_Alphas_ member of the Basis_Alpha_Builder.

void Build_Alphas() Builds the sectors for the model, keeping only the sectors which can produce massless states and placing them into Alpha_Bosons_, Alpha_Fermions_, and Alpha_SUSYs_ as appropriate. It also sets the boolean flag Linearly_Independent_Alphas_. The method begins by declaring a Alpha_Builder using the constructor Alpha_Builder(const vector<Basis_Alpha>& Common_Basis_Alphas, const vector<Basis_Alpha>& Basis_Alphas). It then calls the member function void Build_Alphas() from the Alpha_Builder. It sets Alpha_Bosons_ to the member Alpha_Bosons_ of the Alpha_Builder, Alpha_Fermions_ to the member Alpha_Fermions_ of the Alpha_Builder, and Alpha_SUSYs_ to the member Alpha_SUSYs_ of the Alpha_Builder. After that, Linearly_Independent_Alphas_ is set to the member Linearly_Dependent_Alphas_ of the Alpha_Builder.

void Build_Fermion_Mode_Map() Builds the map between fermion modes for the model. Also sets the boolean flag Consistent_Fermion_Mode_Pairs_. The method begins by creating a Fermion_Mode_Map_Builder object using the default constructor. The member function

void Build_Fermion_Mode_Map(const vector<Basis_Alpha>&

Common_Basis_Alphas) of Fermion_Mode_Map_Builder is called with Common_Basis_Alphas_ as the argument. Then, Fermion_Mode_Map_ is set equal to the member Fermion_Mode_Map_ of Fermion_Mode_Map_Builder. The boolean flag Consistent_Fermion_Mode_Pairs_ is set to the member Consistent_Pairings_ of Fermion_Mode_Map_Builder.

void Build_k_ij() Takes the lower half of the GSO coefficient matrix inputted by the user and builds the upper triangle. Also sets the boolean flag **Consistent_GSO_Matrix_**. The method begins by creating a **GSO_Coefficient_Matrix_Builder** object using the constructor **GSO_Coefficient_Matrix_Builder(const Model& FFHS_Model, const vector<Basis_Alpha>& Common_Basis_Alphas)**, passing it **FFHS_Model_** and **Common_Basis_Alphas_**. Then the member function **void Build_Complete_GSO_Matrix()** of **GSO_Coefficient_Matrix_Builder** is called. Then the **set** method for **k_ij_** is used to put the member **Complete_GSO_Matrix_** of **GSO_Coefficient_Matrix_Builder** into **FFHS_Model_**. Finally, the boolean flag **Consistent_GSO_Matrix_** is set to the member **Consistent_GSO_Matrix_** of the **GSO_Coefficient_Matrix_Builder** object.

void Build_Gauge_Groups() Builds the states from **Alpha_Bosons_** and arranges the positive states into mutually orthogonal sets. These sets form the nonzero positive roots for the gauge groups in the model. Once the sets of nonzero positive roots have been found, the groups are identified and placed in **FFHS_Model_**. The method begins by first building the gauge boson states and placing them in one large list, called **Boson_States**. For each element in **Alpha_Bosons_**, a **State_Builder** object is created with the constructor **State_Builder(const Alpha& The_Alpha, const map<int, int>& Fermion_Mode_Map, const vector<Basis_Alpha>& Common_Basis_Alphas, const GSO_Coefficient_Matrix& k_ij)**, passing it the sector, **Fermion_Mode_Map_**, **Common_Basis_Alphas_**, and **k_ij**. The member function **void Build_States()** of **State_Builder** is then called, which builds the boson states from the boson sector. Those states are then looped over with the intent of adding them to **Boson_States**. Before they are added, the method checks

if the state is positive using the **bool Is_Positive(const map<int, int>& Fermion_Mode_Map)** member function of the **State** class, passing it **Fermion_Mode_Map_**. If the state is positive, then the length squared of the numerator is calculated using the **State** class member function **void Calculate_Length_Squared(const map<int, int>& Fermion_Mode_Map**, passing it **Fermion_Mode_Map_**. Once all this is done (assuming the state is positive), the state is added to **Boson_States**. This process is repeated for all elements of **Boson_Sectors_** so that **Boson_States** has only positive states with **Length_Squared_Numerator_** (a member of **State**) initialized. Now the method groups the states into mutually orthogonal sets. To do this, the method picks a state, pushing it onto a list called **Positive_Roots**. It then computes the gauge charge dot product of that state with each of the states in **Boson_States**, adding any that are nonzero to **Positive_Roots** and erasing them from **Boson_States**. Then, a loop over **Positive_Roots** is started, repeating this process for all of the states in **Positive_Roots**. Since that loop is done via C++ STL **iterator** objects, the loop bounds change as elements are added to **Positive_Roots**. This is done until all of the states left in **Boson_States** are orthogonal to all of the states in **Positive_Roots**. Then, a **Gauge_Group_Identifier** object is created using the constructor **Gauge_Group_Identifier(const list<State>& Positive_Roots, const map<int, int>& Fermion_Mode_Map)**, passing it **Positive_Roots** and **Fermion_Mode_Map_**. A **Gauge_Group** object is then created using the **Get_Group()** member function of **Gauge_Group_Identifier**, which is passed to the **Model** member function **void Add_Gauge_Group(const Gauge_Group& New_Gauge_Group)**. The entire process is repeated until **Boson_States** is empty. A schematic of this process is given in figure A.4.

void Build_SUSY_States() Builds the gravitinos for the model, initializing

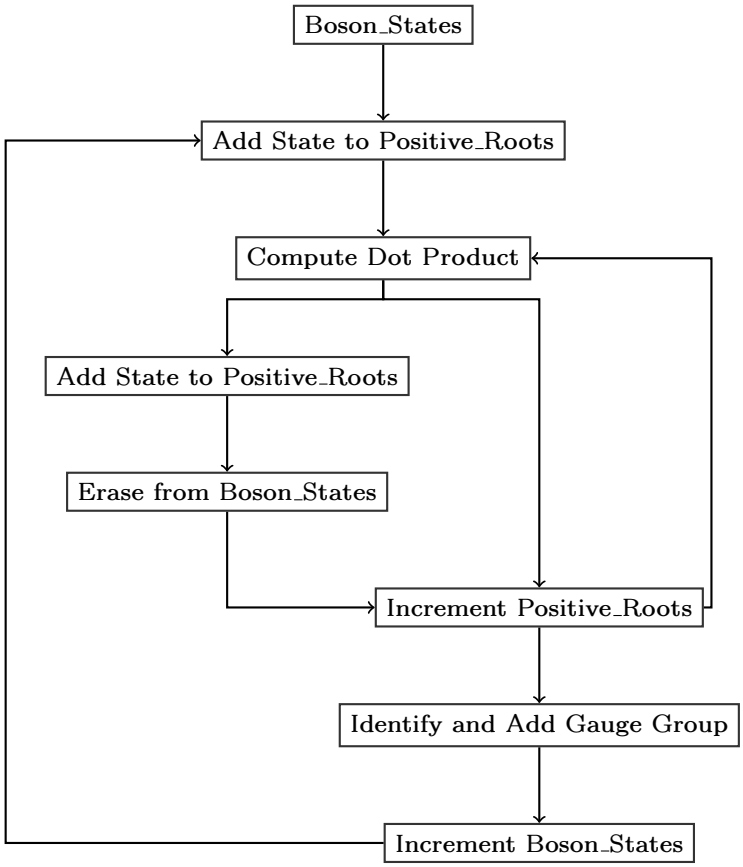


Figure A.4: A schematic of the process used to build the gauge groups.

SUSY_States_. This is done by looping over Alpha_SUSYs_. For each SUSY generating sector, a State_Builder object is created using the constructor State_Builder(const Alpha& The_Alpha, const map<int, int>& Fermion_Mode_Map, const vector<Basis_Alpha>& Common_Basis_Alphas, const GSO_Coefficient_Matrix& k_ij), passing it the SUSY generating sector, Fermion_Mode_Map_, Common_Basis_Alphas_, and k_ij_ (from FFHS_Model_). Then, the State_Builder member function void Build_States() is called. Then, those states are added to SUSY_States_. This process is repeated for the next SUSY generating sector.

void Build_Matter_States() Builds the matter states for the model, initializing Matter_States_ in FFHS_Model_. The method begins by looping over Alpha_Fermions_. For each fermion sector, a State_Builder object is created using the constructor State_Builder(const Alpha& The_Alpha, const map<int, int>& Fermion_Mode_Map, const vector<Basis_Alpha>& Common_Basis_Alphas, const GSO_Coefficient_Matrix& k_ij), passing it the fermion sector, Fermion_Mode_Map_, Common_Basis_Alphas_, and k_ij (in FFHS_Model_). Once the states from the fermion sector are built, the method then checks if the matter state is a supersymmetric partner to a gauge boson using the member function bool Is_SUSY_Partner(const State& New_Matter_State), passing it the state which was created by State_Builder. If the state is not a SUSY partner, then the method must compute the dimensions of the group representations under which the states transform. This is done by looping over the gauge groups and calling int Compute_Rep_Dimension(const State& Weight, const map<int, int>& Fermion_Mode_Map) for each group. Those dimensions are pushed onto a vector. However, since only the highest weight states are needed for the model, if int Compute_Rep_Dimension(const State& Weight, const

map<int, int>& Fermion_Mode_Map) returns 0, then the loop over the gauge groups is broken since the state is not a highest weight in the representation. Note that the special accessor Gauge_Group& rGauge_Group() is used rather than a standard accessor. This is because the Gauge_Group member function int Compute_Rep_Dimension(const State& Weight, const map<int, int>& Fermion_Mode_Map) alters the Gauge_Group object. See the Gauge_Group class documentation for more details. Once the new state has been determined as a non-SUSY partner, highest weight representation for all of the gauge groups, a new Matter_State object is created using the constructor Matter_State(const vector<int>& Numerator, int Denominator, int LM_Size, const vector<int>& Representations), passing it the Numerator_, Denominator_, and LM_Size_ members of the fermion state, as well as the vector containing the representation dimensions. That state is added to Matter_States_ in FFHS_Model_ using void Add_Matter_State(const Matter_State& New_Matter_State). Once this has been done for all of the states produced by all of the sectors, the void Sort_Matter_States() member function of FFHS_Model_ is called to sort the states according to their matter representations.

void Compute_U1_Factors() Computes the number of $U(1)$ factors in the model, setting the U1_Factors_ member of FFHS_Model_. First the maximal rank of the gauge group product making the model is found. This is a function of the number of large space-time dimensions and, consequently, the size of the basis vectors. It is given by the following formula

$$Total\ Rank = \frac{Basis\ Vector\ Size}{4} + 6 \tag{A.24}$$

After that, any rank-cuts made by left-right paired fermion modes are then subtracted from this total. The number of rank-cuts are found

using the **int Find_Rank_Cuts()** function. Then, the total rank of the non-Abelian gauge groups is summed. The number of $U(1)$ factors in a model is the maximal rank (after rank-cuts) minus the total rank of the non-Abelian gauge groups. The **U1_Factors_** member of **FFHS_Model_** is then initialized to that value using it's setter function.

int Gauge_Dot(const State& State1, const State& State2) This method computes the dot product of the gauge charges between **State1** and **State2**. This is done by starting a loop at the first right moving element, and computing the dot product with only the first mode in a complex pair. It returns only the numerator of the dot product. These dot products do not need to be corrected for being in a real basis, since only the first fermion mode in each complex pair is calculated.

bool Is_SUSY_Partner(const State& New_Matter_State) Determines whether **New_Matter_State** is a fermionic superpartner to a gauge boson, returning **true** if it is, and **false** if it is not. This method loops over each of the SUSY states (gravitinos). Nested within that loop, there is a loop over the left movers between **New_Matter_State** and the SUSY state. If the left movers match exactly, then **New_Matter_State** is a SUSY partner, and the method returns **true**. If they do not match, the inner loop is immediately broken and the loop over **SUSY_States_** is incremented. If none of the **SUSY_States_** have a matching left mover to **New_Matter_State**, then **New_Matter_State** is not a SUSY partner, and the method returns **false**.

int Find_Rank_Cuts() Finds and returns the number of rank-cuts a model has. This is done by looping over **Fermion_Mode_Map_** and counting how many left-right pairs there are for the model. The number of rank-cuts for the model is equal to half the number of left-right pairs.

A.2.22 FF_Modular_Invariance_Checker.hh

NAME: Modular_Invariance_Checker

PURPOSE: Holds the functions for checking the modular invariance of a set of basis vectors.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS: None.

CONSTRUCTORS:

Modular_Invariance_Checker() A default constructor. Nothing is initialized, since there are no member variables for this class.

METHODS:

bool Test_Modular_Invariance(const vector<Basis_Alpha>& Basis_Alpha_Set) Checks and returns whether Basis_Alpha_Set has modular invariance, returning true if it does and false if it does not. There are two conditions which must be checked to ensure Basis_Alpha_Set has modular invariance. The first is a set of dot product conditions, checked by bool Check_Dot_Products(const vector<Basis_Alpha>& Basis_Alpha_Set). The other condition is on the number of simultaneous periodic modes, and ensures that the fermion modes can all be paired. This is done with bool Check_Simultaneous_Periodic_Modes(const vector<Basis_Alpha>& Basis_Alpha_Set). Both of these functions must return true for Basis_Alpha_Set to have modular invariance.

bool Check_Dot_Products(const vector<Basis_Alpha>& Basis_Alpha_Set) Checks the dot product conditions required for modular invariance for Basis_Alpha_Set.

Those equations are

$$N_i\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B = 0 \pmod{16} \text{ (even order)}, \quad \text{(A.25)}$$

$$N_i\vec{\alpha}_i^B \cdot \vec{\alpha}_i^B = 0 \pmod{8} \text{ (odd order)}, \quad \text{(A.26)}$$

$$LCM(N_i, N_j)\vec{\alpha}_i^B \cdot \vec{\alpha}_j^B = 0 \pmod{8}, \quad \text{(A.27)}$$

where N_i is the order, $LCM(N_i, N_j)$ is the least common multiple between N_i and N_j, and $\vec{\alpha}_i^B$ is the basis alpha for the model. The method loops over **Basis_Alpha_Set**, testing all of the dot products for each of its elements. If any of them fail, the function immediately returns **false**. Otherwise, after the loops are completed, the method returns **true**.

bool Check_Simultaneous_Periodic_Modes(const vector<Basis_Alpha>&

Basis_Alpha_Set) Tests the condition that the number of simultaneous periodic modes for any three $\vec{\alpha}^B$'s be even. This condition ensures that the fermion modes can all be paired. The method does this by looping over all three element combinations of **Basis_Alpha_Set**. The method adds each set of modes together from the three basis alphas. If that sum is equal to the sum of the denominators of those basis alphas, then all three modes were periodic. The method counts how many times this occurs, and if it is ever an odd number, immediately returns **false**. If the loops complete without exiting, the method returns **true**.

A.2.23 FF_State.hh

NAME: State

PURPOSE: Holds the information needed to specify and operate with a state in the model.

OBJECTS CREATED BY: State_Builder

USED IN: State_Builder, GSO_Projector, Matter_State, Model_Builder, Model, Gauge_Group, Gauge_Group_Identifier

MEMBERS:

vector<int> Numerator_ Holds the numerator for the state.

int Denominator_ Holds the denominator for the state.

int LM_Size_ Holds the size of the left movers for the state.

int Length_Squared_Numerator_ The numerator for the length squared of the state.

int Length_Squared_Denominator_ The denominator for the length squared of the state.

CONSTRUCTORS:

State() The default constructor, which does not initialize any member variables. Not recommended for explicit use.

State(const vector<int>& Numerator, int Denominator, int LM_Size) Initializes Numerator_ to Numerator, Denominator_ to Denominator, and LM_Size_ to LM_Size.

METHODS:

void Calculate_Length_Squared(const map<int, int>& Fermion_Mode_Map)

Computes the length squared of the gauge charges by taking Fermion_Mode_Map and looping over the right moving modes which are the first element in a complex pair, squaring each mode's value. Initializes Length_Squared_Numerator_ to the numerator of the dot product, and Length_Squared_Denominator_ to the denominator of the dot product.

bool Is_Positive(const map<int, int>& Fermion_Mode_Map) Takes the map

between complex fermion modes and determines if the state is positive in the gauge charge space. This is done by examining the first nonzero right moving mode that is also the first element in a complex pair (i.e. the first nonzero gauge charge). If that charge is positive, the method returns **true**, otherwise it returns **false**.

virtual bool operator==(const State& State2) const Returns **true** if and only if the numerators of ***this** and **State2** are exactly equal. This method could be improved by making it a **friend** function rather than a member function.

virtual bool operator != (const State& State2) const Returns the opposite of **virtual bool operator==(const State& State2)**. This method could be improved by making it a **friend** function rather than a member function.

virtual bool operator<(const State& State2) Returns **true** if the numerator of ***this** is less than the numerator of **State2**, and returns **false** otherwise. This method could be improved by making it a **friend** rather than a member function.

A.2.24 FF_State_Builder.hh

NAME: State_Builder

PURPOSE: Builds the physically consistent massless states from a specified sector for a model. First builds all massless states, then performs the GSO projections.

OBJECTS CREATED BY: Model_Builder

USED IN: Model_Builder

MEMBERS:

Alpha The_Alpha_ Holds the sector which produces the states. Since this member is initialized by reference in the constructor, it could be any of Alpha's subclasses: Alpha_Fermion, Alpha_Boson, or Alpha_SUSY.

char Alpha_Type_ Holds a character indicative of the type of sector that The_Alpha_ is. Since the constructor takes a reference to an Alpha object, the subclasses of Alpha can also be initialized to The_Alpha_. The Type_ member of that object is saved in this variable when the constructor is called.

map<int, int> Fermion_Mode_Map_ Holds the fermion mode pairings. Initialized in constructor.

GSO_Projector GSO_ Handler object which performs the GSO projections on the massless states once they are created. Initialized in constructor.

list<State> States_ The massless states produced by The_Alpha_ which passed the GSO projections.

CONSTRUCTORS:

State_Builder(const Alpha& The_Alpha, const map<int, int>& Fermion_Mode_Map,

const vector<Basis_Alpha>& Common_Basis_Alphas,

const GSO_Coefficient_Matrix& k_ij) The method first initializes The_Alpha_ to The_Alpha. Because this is an argument by reference, it can (and is intended to) take any of Alpha's subclasses. The Type_ member of Alpha and its subclasses indicates whether it is an Alpha_Fermion, Alpha_Boson, or Alpha_SUSY. That member is stored in Alpha_Type_. Fermion_Mode_Map_ is initialized to Fermion_Mode_Map, and a GSO_Projector object is created using the constructor GSO_Projector(const vector

<Basis_Alpha> Common_Basis_Alphas, const Alpha& The_Alpha,

const GSO_Coefficient_Matrix& k_ij, const map<int, int>& Fermion_Mode_Map).

Then GSO_ is initialized to that object.

METHODS:

void Build_States() Interface function which selects the helper to call for building the states. The method first determines which type of state to build based on **Alpha_Type_**. This is done with a **switch** statement on that member. If **Alpha_Type_** is '**b**', then **void Build_Boson_States()** is called. If **Alpha_Type_** is '**f**', then **void Build_Fermion_States()** is called. Finally, if **Alpha_Type_** is '**s**', then **void Build_SUSY_States()** is called.

void Display_The_Alpha() const Calls the **void Display() const** method for **The_Alpha_**. For debugging.

void Display_States() const Loops over **States_** calling each elements **void Display() const** function. For debugging.

void Build_Fermion_States() Builds the fermion states for **The_Alpha_** that are massless and consistent with the GSO projections. Initializes **States_**. This method begins by calculating the number of large space-time dimensions, calling **int Calculate_Large_ST_Dimensions(int LM_Size)**, passing it **The_Alpha_**'s **LM_Size_** member. After that, a **State_LM_Builder** object is created using the constructor **State_LM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions, const map<int, int>& Fermion_Mode_Map)**, passing it **The_Alpha_**, the previously calculated number of large space-time dimensions, and **Fermion_Mode_Map_**. A **State_RM_Builder** object is also created using the constructor **State_RM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions, const map<int, int>& Fermion_Mode_Map)**, passing it **The_Alpha_**, the previously calculated number of large space-time dimen-

sions, and Fermion_Mode_Map_. Then, the massless left and right moving modes of the states are built calling void Build_Massless_State_LMs() from the State_LM_Builder object and void Build_Massless_State_RMs() from the State_RM_Builder object. Then, the Massless_State_LMs_ member of State_LM_Builder is looped over, and within that loop the Massless_State_RMs_ member of State_RM_Builder is looped over, merging the left and right movers to form all possible complete states. The actual new State objects are created by merging the numerators (pulled from State_LM_Builder and State_RM_Builder), then calling the constructor State(const vector<int>& Numerator, int Denominator, int LM_Size), passing it the combined numerator, twice the denominator for The_Alpha_ (recall the state charge values are half that of the sector phase values), and the LM_Size_ member of The_Alpha_. For each complete state that is formed, that state is passed to bool GSOP(const State& New_State). If that function returns true, that state is added to States_. If it does not, the state is not added. When this method finishes, the member States_ contains only massless states produced by The_Alpha_ which have passed the GSO projections.

void Build_Boson_States() Builds the boson states from The_Alpha_ which are massless and consistent with the GSO projections. Initializes States_. The method begins by calculating the number of large space-time dimensions using the function int Calculate_Large_ST_Dimensions(int LM_Size), passing it LM_Size_ of The_Alpha_. Then, a State_RM_Builder is created using the constructor State_RM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions, const map<int, int>& Fermion_Mode_Map), passing it The_Alpha_, the previously calculated number of large space-time dimensions, and Fermion_Mode_Map_. To build the boson state right movers,

the State_RM_Builder method void Build_State_RMs() is called. Those right movers are then looped over. Each right moving boson state is appended to the boson left mover, which has 1's for the first complex space-time pair, and 0's for all other left moving fermion modes. Once the complete boson state is created (using the constructor State(const vector<int>& Numerator, int Denominator, int LM_Size), passing it the complete numerator, twice the denominator of The_Alpha_ since a state has half the phase values of the sector, and the LM_Size_ member of The_Alpha_), the method calls bool GSOP(const State& New_State) from GSO_ to see if the state passes the GSO projections. If that method returns true, the the boson state is added to States_. If the method returns false, the state is not added.

void Build_SUSY_States() Builds the gravitino states from The_Alpha_ that are massless and consistent with the GSO projections. Initializes States_. The method begins by calculating the number of large space-time dimensions using int Calculate_Large_ST_Dimensions(int LM_Size). Then, a State_LM_Builder object is created using the constructor

State_LM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions, const map<int, int>& Fermion_Mode_Map), passing it The_Alpha_, the previously calculated large space-time dimensions, and Fermion_Mode_Map_. Then, the massless left movers are created using the

void Build_Massless_State_LMs() member function of State_LM_Builder.

Those left movers are looped over, and the right mover (all 0's) for gravitinos is added to them. The new state is created by calling the State constructor State(const vector<int>& Numerator, int Denominator, int LM_Size), passing it the newly created numerator, twice the denominator of The_Alpha_ since the state charge elements are half the sector

phase values, and LM_Size_ of The_Alpha_. Once the gravitino state has been created, the method then calls **bool GSOP(const State& New_State)** from GSO_ to see if the gravitino passes the GSO projections. If the state passes, it is added to States_, otherwise, it is not.

int Calculate_Large_ST_Dimensions(int LM_Size) Calculates and returns the number of large space-time dimensions given the size of the left mover for a state, basis vector, or sector. Uses the formula A.8.

A.2.25 FF_State_LM_Builder.hh

NAME: State_LM_Builder

PURPOSE: Builds the massless left moving parts of the states produced by a given fermion or SUSY sector.

OBJECTS CREATED BY: State_Builder

USED IN: State_Builder

MEMBERS:

int Large_ST_Dimensions_ The number of large space-time dimensions for the model.

vector<int> Alpha_LM_Numerator_ The left moving part of the numerator for the sector producing the state left movers.

int Alpha_LM_Denominator_ The denominator of the sector producing the state left movers.

map<int, int> Fermion_Mode_Map_ The map of complex fermion modes for the model.

list<vector<int> > ST_LM_Modes_ The left moving space-time modes.

list<vector<int> > Compact_LM_Modes_ The left moving compact modes.

list<vector<int> > Massless_State_LMs_ All the massless left movers.

CONSTRUCTORS:

State_LM_Builder(const vector<int>& Alpha_LM_Numerator,

int Alpha_LM_Denominator, int Large_ST_Dimensions,

const map<int, int>& Fermion_Mode_Map) Initializes Alpha_LM_Numerator_ to Alpha_LM_Numerator, Alpha_LM_Denominator_ to Alpha_LM_Denominator, Large_ST_Dimensions_ to Large_ST_Dimensions, and Fermion_Mode_Map_ to Fermion_Mode_Map.

State_LM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions,

const map<int, int>& Fermion_Mode_Map) Initializes Alpha_LM_Numerator_ to the left moving part of Numerator_ in The_Alpha, and Alpha_LM_Denominator_ to Denominator_ in The_Alpha. Also initializes Large_ST_Dimensions_ to Large_ST_Dimensions, and Fermion_Mode_Map_ to Fermion_Mode_Map.

METHODS:

void Build_Massless_State_LMs() Builds the massless left movers. Initializes Massless_State_LMs_. The method begins by creating an appropriately sized empty vector to serve as an initial parameter for void Build_Compact_LM_Modes(int element, vector<int>& LM_Compact). The method then calls void Build_ST_LM_Modes() and void Build_Compact_LM_Modes(int element, vector<int>& LM_Compact), passing the latter function 0 and the empty vector. After that, the method produces all possible combinations between ST_LM_Modes_ and Compact_LM_Modes_, pushing each combination onto Massless_State_LMs_. There is an optimization which pushes only the space-time modes onto Massless_State_LMs_ if there are ten large space-time dimensions, since there are not compact modes in that case.

void Display_Massless_State_LMs() const Displays the elements of Massless_State_LMs_ onto the screen for debugging.

void Display_Fermion_Mode_Map() const Displays Fermion_Mode_Map_ onto the screen for debugging.

void Build_ST_LM_Modes() Builds the space-time left moving fermion modes for a given model, initializing ST_LM_Modes_. For models with less space-time dimensions, this is simply adding the $\frac{1}{2}$ modes since the GSOPs only allow one parity (selected by the space-time fermion modes and x modes) to survive for those models. For models with ten space-time dimensions, even and odd parities can be present in a model, so in addition to the vector of eight real $\frac{1}{2}$ modes there is also a vector of six real $\frac{1}{2}$ modes and two real $-\frac{1}{2}$ modes.

void Build_Compact_LM_Modes(int element, vector<int>& LM_Compact) Builds the compact fermion modes for a given model, initializing Compact_LM_Modes_. Takes the element to be lowered and the current compact left moving modes as parameters. This method uses recursion to apply the lowering operator to all nonzero modes of the compact left moving elements. The method begins by checking to see if the parameter **element** is less than the size of LM_Compact. If it is, then the current loop in the recursion has terminated, and LM_Compact is pushed onto Compact_LM_Modes_. Otherwise, the method then checks to see if the following conditions are true:

- The left moving compact mode at **element** is nonzero.
- The left moving compact mode at **element** is not the second in a complex pair.

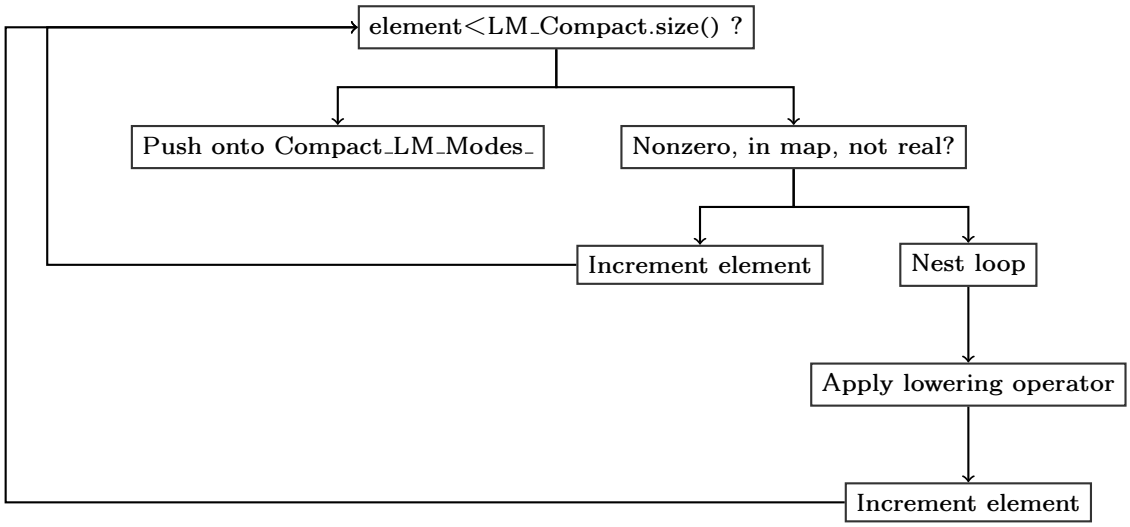


Figure A.5: A schematic of the recursive algorithm used to apply the $\vec{F}$ operator to the LM of a state.

- The left moving compact mode at **element** is not the first element of a real left-right pair. This is checked using the function **bool Real_LM_Mode(int element)**, passing it **element**.

If any of these are false then the method calls itself, passing it **element+1** and **LM_Compact**. If they are all true, then the method starts a loop over the possible lowering operator values (0 or -1). Since the numerator is being operated upon, the method subtracts the denominator from the elements. This is done for the fermion mode at **element** and it's complex partner. Once the loop has been started and the values adjusted, the method calls itself, passing **element+1** and **LM_Compact**. This process is shown schematically in Figure A.5.

bool Real_LM_Mode(int element) Takes the index of a compact left moving element and returns whether it is a real fermion mode or not. The method begins by adjusting the value of **element** to fit the full state vector. This is necessary because it is called within **void Build_Compact_LM_Modes(int element, vector<int>& LM_Compact)**, which only carries the compact left

moving modes. Once the value of **element** is adjusted, **Fermion_Mode_Map_** is checked. If the "complex" partner of the mode indexed by **element** is a right moving mode, the method returns **true**. Otherwise it returns **false**.

bool In_Map(int element) Takes the index of a compact left moving element and returns whether it is in **Fermion_Mode_Map_** as a key or a value. If it is a key, then the element is the first mode in a complex pair, otherwise it is the second element in the pair. The method must first adjust the value of **element** to fit the full state vector. This is necessary because it is called within **void Build_Compact_LM_Modes(int element, vector<int>& LM_Compact)**, which only carries the compact left moving modes. Once the value of element is adjusted, the **find** function of the C++ STL **map** is called, passing it the adjusted **element**. If it returns anything other than **Fermion_Mode_Map_.end()**, the method returns **true**. Otherwise, it returns **false**.

A.2.26 FF_State_RM_Builder.hh

NAME: State_RM_Builder

PURPOSE: Builds the massless right movers for states coming from a boson or fermion sector.

OBJECTS CREATED BY: State_Builder

USED IN: State_Builder

MEMBERS:

int Large_ST_Dimensions_ The number of large space-time dimensions for the model.

vector<int> Alpha_RM_Numerator_ The numerator of the right moving part of the sector which is generating the states.

int Alpha_RM_Denominator_ The denominator of the right moving part of the sector which is generating the states.

map<int, int> Fermion_Mode_Map_ The map of complex fermion modes for the model.

int LM_Size_ The number of left moving modes for the sector producing the states.

int Mass_Limit_ The maximal mass squared value for the right movers.

map<int, int> Reverse_Fermion_Mode_Map_ The reverse map for Fermion_Mode_Map_.

list<vector<int> > Massless_State_RMs_ The massless right movers for the states.

CONSTRUCTORS:

State_RM_Builder(const vector<int>& Alpha_RM_Numerator, int Alpha_RM_Denominator, int Large_ST_Dimensions, const map<int, int>& Fermion_Mode_Map) Initializes Alpha_RM_Numerator_ to Alpha_RM_Numerator, Alpha_RM_Denominator_ to Alpha_RM_Denominator, Large_ST_Dimensions_ to Large_ST_Dimensions, and Fermion_Mode_Map_ to Fermion_Mode_Map. It then calls void Build_Reverse_Fermion_Mode_Map(), which initializes Reverse_Fermion_Mode_Map_. After that, the constructor initializes LM_Size_ to its value using equation A.1. Finally, Mass_Limit_ is initialized according to the equation

$$\text{Mass Limit} = 2\times 2\times \text{Denominator}\times 2\times \text{Denominator} = 2\times 4\times \text{Denominator}^2 \tag{A.28}$$

State_RM_Builder(const Alpha& The_Alpha, int Large_ST_Dimensions,

const map<int, int>& Fermion_Mode_Map) This constructor begins by stripping the right mover out of Numerator_ in The_Alpha, using it to initialize Alpha_RM_Numerator_. It uses Denominator_ from The_Alpha to initialize Alpha_RM_Denominator_. It then initializes Large_ST_Dimensions_ to Large_ST_Dimensions and Fermion_Mode_Map_ to Fermion_Mode_Map. It also initializes LM_Size_ to LM_Size_ of The_Alpha. Then, Reverse_Fermion_Mode_Map_ is initialized by calling void Build_Reverse_Fermion_Mode_Map(). Finally, Mass_Limit_ is initialized via the equation A.28.

METHODS:

void Build_Massless_State_RMs() Interface function which starts the recursive algorithms for building the state right movers. Begins by creating an unraised, unlowered right mover (set equal to the Alpha_RM_Numerator_), then starts the recursion with void Select_F_Operator(int element, vector<int>& RM, int Mass), passing it a 0, the vector set equal to Alpha_RM_Numerator_, and the mass of Alpha_RM_Numerator_, calculated by int Compute_Mass(vector<int>& RM).

void Display_Massless_State_RMs() const Prints the elements of Massless_State_RMs_ onto the screen for debugging.

void Build_Reverse_Fermion_Mode_Map() Builds the reverse map to Fermion_Mode_Map_, initializing Reverse_Fermion_Mode_Map_. This method should be called after the member Fermion_Mode_Map_ has been initialized, otherwise an exception will be thrown. This method could be improved by allowing it to take the fermion mode map as an argument rather than calling the member Fermion_Mode_Map_.

bool In_Map(int element) Checks whether **element** is in **Fermion_Mode_Map_** as a key by calling the C++ STL **map** function **find** function. Note that **element** must be shifted, since **element** is the index of the mode in **Alpha_RM_Numerator_**, while **Fermion_Mode_Map_** has the mode indices for the entire state vector. Returns **true** if the mode is found, returns **false** if it isn't.

bool Real_RM_Mode(int element) Takes the index of an element and determines whether it is the right moving part of a left-right pair, returning **true** if it is, and **false** if it is not. This is checked by looking for **element** in **Reverse_Fermion_Mode_Map_** using the C++ STL map **find** function. It is important to note that **element** needs to be adjusted, since it is the index **element** is an index for an element of **Alpha_RM_Numerator_**, and **Fermion_Mode_Map_** holds the indices for the full state rather than just the right mover.

void Select_F_Operator(int element, vector<int>& RM, int Mass) This method selects whether a raising and lowering operator, a raising operator only, a lowering operator only, or no operator should be applied to the element indexed by **element** of **RM**. The method also accepts the mass of **RM** as a passed parameter rather than calculating it each time to save computing power. It begins by checking if **element** is at the end of **RM**. If it is, then the recursive nesting is finished, and **RM** is pushed onto **Massless_State_RMs_** if it is massless (that is, if **Mass** is equal to **Mass_Limit_**). If **element** is not at the end of **RM**, the method checks to see which type of mode it is. If the mode indexed by **element** is the second in a complex right moving pair, then **element** is incremented and the method is called again, passing it the incremented **element**, **RM**, and **Mass**. If the mode indexed by **element** is the second in a

real pair, then that mode should be lowered only. The method first checks whether the mode will become massive when lowered by calling **int Mass_Increase_Lower(int element)** and adding it to **Mass**. If the sum is less than **Mass_Limit_**, the state will not become massive when lowered, and **void Apply_Real_F_Operator(int element, vector<int> RM, int Mass)** is called, passing it **element**, **RM**, and **Mass**. If the state will become massive if the lowering operator is applied, then **element** is incremented and the function is called recursively, passing it the incremented **element**, **RM**, and **Mass**. If the mode is the first in a complex pair, then the method checks whether the state will become massive if a raising and a lowering operator is applied. This is done by calling **int Mass_Increase_Lower(int element)** and **int Mass_Increase_Raise(int element)**, passing both of these **element**. Those values are added to **Mass** (separately) and checked if they make the state massive. If the state becomes massive when raised or lowered, then **element** is incremented and the function is called recursively, passing it the incremented **element**, **RM**, and **Mass**. If the state becomes massive only when raised, but not lowered, then **void Apply_Complex_F_Operator(int element, vector<int> RM, int F_Lower, int F_Raise, int Mass)** is called, passing it **element**, **RM**, **-1**, **0**, and **Mass**. If the state becomes massive only when lowered, then **void Apply_Complex_F_Operator(int element, vector<int> RM, int F_Lower, int F_Raise, int Mass)** is called, passing it **element**, **RM**, **0**, **1**, and **Mass**. If the state does not become massive when either the raising or lowering operator is called, then the method calls **void Apply_Complex_F_Operator(int element, vector<int> RM, int F_Lower, int F_Raise, int Mass)**, passing it **element**, **RM**, **-1**, **1**, and **Mass**. The process is shown schematically in figure A.6.

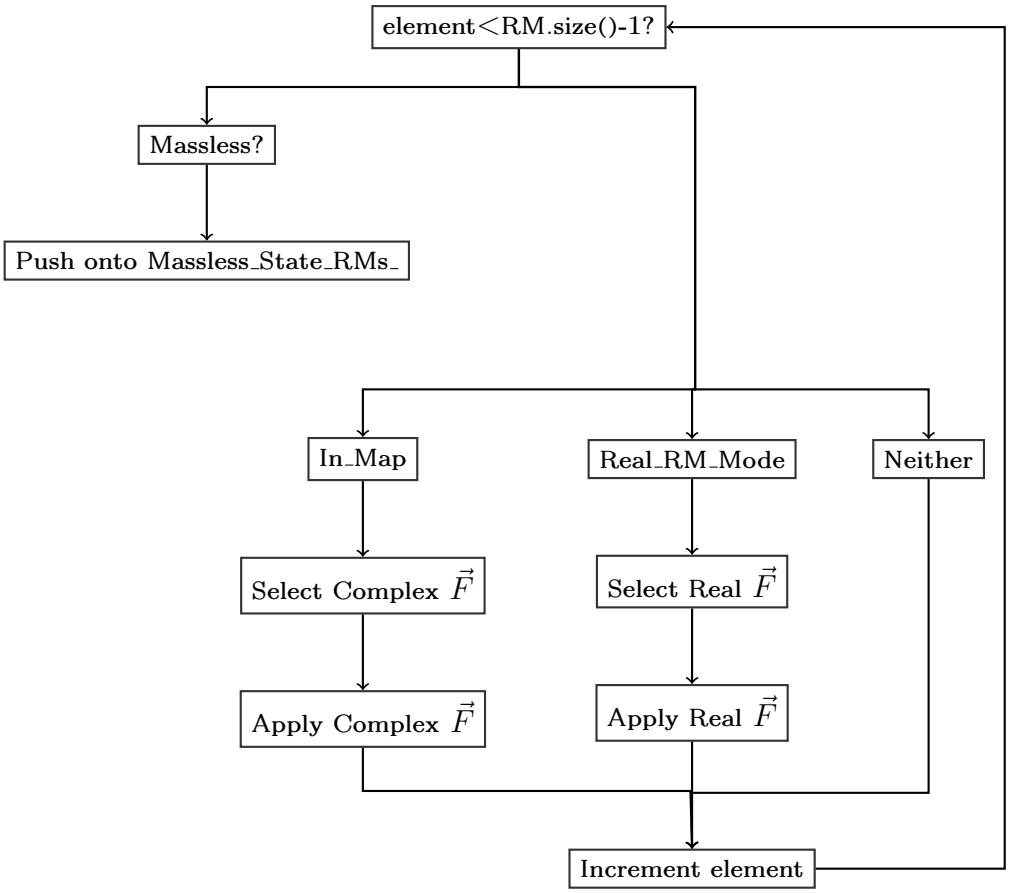


Figure A.6: A schematic for the algorithm which selects $\vec{F}$ for a state's RM.

int Compute_Mass(vector<int>& RM) Takes a state right mover (RM) and computes the mass squared, returning the numerator of that product. It loops over the modes in RM, checking to see if they are real using **bool Real_RM_Mode(int element)**, passing it the loop index for **element**. It also checks to see if the mode is in the map using **bool In_Map(int element)**, also passing it the loop index for **element**. If the mode is in **Fermion_Mode_Map_**, then the mass squared is incremented by the square of the mode. If the mode is a real mode and is a twisted mode (the absolute value of the mode is equal to **Alpha_RM_Denominator_**), then the mass squared is incremented by half the square of the mode. If the mode is a real mode but is untwisted, then the mass squared is incremented by the square of the mode. The special treatment is needed due to a redundancy in the real modes' lowering operator - the mass increase of real modes is doubled. However, lowering real modes with twists of $\frac{1}{2}$ does not increase the mass, so only the contribution of the original mode is counted. In the case of a lowering operator applied to an untwisted mode, the mass increase is doubled, and the contribution of those modes to the mass squared is weighed equivalently to the complex right moving modes. Once the mass squared has been calculated, the method returns the numerator of the mass squared.

int Mass_Increase_Raise(int element) Takes the index of a right moving mode and computes the mass increase that would occur should the raising operator be applied. It returns the value of the equation

$$\text{Mass\_Increase} = 4 \times \text{Denom}^2 + 4 \times \text{Denom} \times \text{Num.at(element)}, \quad \text{(A.29)}$$

where Denom is the denominator of the sector producing the states and Num is the numerator of the sector producing the states.

int Mass_Increase_Lower(int element) Takes the index of a right moving mode and computes the mass increase that would occur should the lowering operator be applied. It returns the value of the equation

$$\text{Mass\_Increase} = 4 \times \text{Denom}^2 - 4 \times \text{Denom} \times \text{Num.at(element)}, \quad \text{(A.30)}$$

where Denom is the denominator of the sector producing the states and Num is the numerator of the sector producing the states.

void Apply_Real_F_Operator(int element, vector<int> RM, int Mass) Nests a loop in which an $\vec{F}$ operator is applied to a real right moving element. This is different from a complex element for two reasons. Firstly, only the lowering operator is applied. A real mode with a raised element is identical to one which is lowered, so the raising operator does not produce a different state. This redundancy requires corrections which must be applied to the mass squared and GSO projection calculations, and are accounted for elsewhere. Secondly, there is not a complex partner which needs to be operated upon. The left moving mode in a real pair does not get the $\vec{F}$ operator treatment. This is another redundancy present in the construction which must be accounted for in the GSO projections and the mass squared calculations. The method calls a loop between -1 and 0 (representing lowering and not lowering). Within that loop, the operator is applied to the element of RM, the the change in mass is calculated using a modified version of equation (A.30) which only adds to the mass when the loop index is nonzero. Then the method increments **element**, passing it, the new RM, and the new **Mass** to **void Select_F_Operator(int element, vector<int>& RM, int Mass)**.

void Apply_Complex_F_Operator(int element, vector<int> RM, int F_Lower,

int F_Raise, int Mass) Nests a loop in which an $\vec{F}$ operator is applied to a complex right moving element. The bounds of the loop are set by **F_Lower** and **F_Raise**, which determine whether $\vec{F}$ will only lower, only raise, or raise and lower the **element**'th mode of **RM**. **Mass**, the mass squared of **RM**, is also needed, as it will be changed when $\vec{F}$ is applied. The method first applies $\vec{F}$ to the mode of **RM** indexed by **element** as well as it's complex partner, which is stored in **Fermion_Mode_Map_**. Then the increase in mass is calculated using a modified version of equations (A.29, A.30) which only adds to the mass when the loop index is nonzero. Then the method increments **element** passing it, the new **RM**, and the new **Mass** to **void Select_F_Operator(int element, vector<int>& Mass)**.

A.3 FF Framework Class Inheritance Structure

Presented here are the classes which use inheritance. Each derived class inherits all members of the base class, and any added members or differences are detailed.

A.3.1 **Alpha** *Class Inheritance*

Below is a description of the classes inheriting from **Basis_Alpha**, and what is changed or added with each inheriting class. A graphical representation is shown in figure A.7.

Basis_Alpha The base class for this inheritance tree.

Alpha Inherits from **Basis_Alpha**, adding **bool operator<(const Alpha& Other_Alpha)**, **int Mass_Left()**, **int Mass_Right()**, **virtual bool Type_**, and **vector<int> Coefficients_**.

Alpha_Fermion Inherits from **Alpha**. Changes **bool Type_** to '**f**'.

Alpha_Boson Inherits from **Alpha**. Changes **bool Type_** to '**b**'.

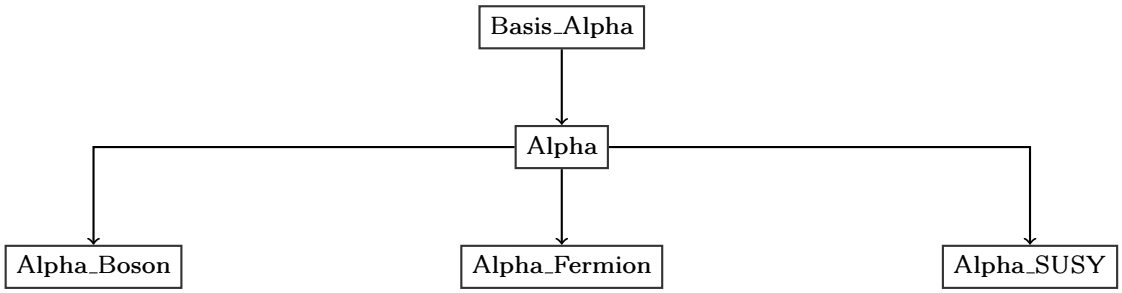


Figure A.7: The inheritances of the **Alpha**-type classes.

Alpha_SUSY Inherits from **Alpha**. Changes **bool Type_** to '**s**'.

A.3.2 **State** *Class Inheritance*

Below is a description of the inheritance structure of the **State** and **Matter_State** classes, as well as descriptions of what is changed from **State** to **Matter_State**.

State The base class for this inheritance tree.

Matter_State Modifies **bool operator<(const Matter_State& Matter_State2) const** and **bool operator==(const Matter_State& Matter_State2) const**. Also adds **vector<int> Representations_**.

A.4 Using the Makefile

This section describes the directory structure in the FF Framework, as well as the necessary requirements needed to operate the included makefile, called **Makefile**.

A.4.1 Directory Structure

The directory structure of the FF Framework is as follows.

FF_Framework/ Contains the headers (.**hh**) for the FF Framework classes.

src/ Contains the source files (.**cpp**) for the FF Framework classes.

obj/ Contains the object files (.**o**) for the FF Framework classes.

ana/ Contains the sources and object files for analysis and builder programs which use the classes of the FF Framework. Any source to be compiled into an executable using the FF Framework should be placed here.

There can be other directories as well, such as those for data sets or tex files. However, it is not recommended those be kept under version control, as the files can get quite large.

A.4.2 Creating Executables

To use the makefile, have a source ready to compile in **ana/**. The command to compile the source into an executable is the root name of the source with the extension **.exe** added. For example, if the source were named **NAHE_Builder.cpp**, the following command will compile the source into an executable when given in the FF Framework's root directory.

make NAHE_Builder.exe

The makefile looks for all headers in **FF_Framework/** and all sources in **src/**. It then turns any classes whose source files have changed into object files, placing them in **obj/**. Next, it compiles the executable itself, placing the object file for the executable in **ana/** and linking everything from **obj/** to create the **.exe** file, which is placed in the root directory for the framework.

The makefile will scan the entire **FF_Framework/** and **src/** directories, so adding classes to the framework for analysis or extra functionality do not require makefile modification, unless they are placed in different directories than those specified above.

A.4.3 Debugging and Optimization

The **g++** compiler has flags for optimization and debugging, and these can be set inside the makefile for the FF Framework. However, they require the makefile

itself to be altered, so it is recommended that a **clean** command be called so that all sources will be compiled from scratch.

The variable controlling the optimization flag is OPT. It tells **make** whether to turn on the **-O3** compiler flag for **g++**. To enable compiler optimization, set this variable to YES. To disable it, set it to NO.

The variable controlling the debugging flag is DBG. It tells **make** whether to turn on the **-g** compiler flag for **g++**. To enable debugging, set this variable to YES. To disable it, set it to NO. It is not recommended that both of these flags be simultaneously used. The **gdb** debugger will not be as informative due to the compiler optimizations.

A.4.4 Other Makefile Functions

There are other functions that are defined by the makefile for convenience. The first is the **clean** function, which deletes the contents of **obj** as well as any **.o** files in **ana/** and any **.exe** files in the root directory for the FF Framework. This is useful if header files or compiler flags are changed. The other function is **echo**, which outputs the files that **make** uses and the variables under which they are stored. This is useful if **make** is, for whatever reason, not functioning properly.

BIBLIOGRAPHY

[1] K. Nakamura et al., "Review of particle physics," J. Phys. **G37**, 075021 (2010).

[2] B. Zwiebach, *A First Course in String Theory*, Cambridge University Press, 2004.

[3] E. Kiritsis, *String Theory in a Nutshell*, Princeton University Press, 2007.

[4] J. H. S. Katrin Becker, Melanie Becker, *String Theory and M-Theory: A Modern Introduction*, Cambridge University Press, 2007.

[5] M. Gasperini, *Elements of String Cosmology*, Cambridge University Press, 2007.

[6] I. Antoniadis, C. P. Bachas, and C. Kounnas, "Four-Dimensional Superstrings," Nucl. Phys. **B289**, 87 (1987).

[7] I. Antoniadis and C. Bachas, "4-D Fermionic Superstrings with Arbitrary Twists," Nucl. Phys. **B298**, 586 (1988).

[8] H. Kawai, D. C. Lewellen, and S. H. H. Tye, "Construction of Fermionic String Models in Four- Dimensions," Nucl. Phys. **B288**, 1 (1987).

[9] J. Bagger, D. Nemeschansky, N. Seiberg, and S. Yankielowicz, "Bosons, Fermions and Thirring Strings," Nucl. Phys. **B289**, 53 (1987).

[10] H. K. Dreiner, J. L. Lopez, D. V. Nanopoulos, and D. B. Reiss, "String Model Building in the Free Fermionic Formulation," Nucl. Phys. **B320**, 401 (1989).

[11] R. Bousso and J. Polchinski, "Quantization of four-form fluxes and dynamical neutralization of the cosmological constant," JHEP **06**, 006 (2000).

[12] S. Ashok and M. R. Douglas, "Counting flux vacua," JHEP **01**, 060 (2004).

[13] K. R. Dienes, "Statistics on the heterotic landscape: Gauge groups and cosmological constants of four-dimensional heterotic strings," Phys. Rev. **D73**, 106010 (2006).

[14] K. R. Dienes, M. Lennek, D. Senechal, and V. Wasnik, "Supersymmetry versus Gauge Symmetry on the Heterotic Landscape," Phys. Rev. **D75**, 126005 (2007).

[15] K. R. Dienes and M. Lennek, "Correlation Classes on the Landscape: To What Extent is String Theory Predictive?," Phys. Rev. **D80**, 106003 (2009).

[16] B. Assel, K. Christodoulides, A. E. Faraggi, C. Kounnas, and J. Rizos, "Classification of Heterotic Pati-Salam Models," (2010).

[17] K. R. Dienes and M. Lennek, "Fighting the floating correlations: Expectations and complications in extracting statistical correlations from the string theory landscape," Phys. Rev. **D75**, 026008 (2007).

[18] G. B. Cleaver, A. E. Faraggi, D. V. Nanopoulos, and J. W. Walker, "Phenomenological study of a minimal superstring standard model," Nucl. Phys. **B593**, 471 (2001).

[19] J. L. Lopez, D. V. Nanopoulos, and K.-j. Yuan, "The Search for a realistic flipped SU(5) string model," Nucl. Phys. **B399**, 654 (1993).

[20] A. E. Faraggi, D. V. Nanopoulos, and K.-j. Yuan, "A Standard Like Model in the 4D Free Fermionic String Formulation," Nucl. Phys. **B335**, 347 (1990).

[21] A. E. Faraggi, "Construction of realistic standard - like models in the free fermionic superstring formulation," Nucl. Phys. **B387**, 239 (1992).

[22] I. Antoniadis, G. K. Leontaris, and J. Rizos, "A Three generation SU(4) x O(4) string model," Phys. Lett. **B245**, 161 (1990).

[23] G. K. Leontaris and J. Rizos, "N=1 supersymmetric SU(4)xSU(2)LxSU(2)R effective theory from the weakly coupled heterotic superstring," Nucl. Phys. **B554**, 3 (1999).

[24] A. E. Faraggi, "A New standard - like model in the four-dimensional free fermionic string formulation," Phys. Lett. **B278**, 131 (1992).

[25] A. E. Faraggi, "Aspects of nonrenormalizable terms in a superstring derived standard - like Model," Nucl. Phys. **B403**, 101 (1993).

[26] A. E. Faraggi, "Generation mass hierarchy in superstring derived models," Nucl. Phys. **B407**, 57 (1993).

[27] A. E. Faraggi, "Hierarchical top - bottom mass relation in a superstring derived standard - like model," Phys. Lett. **B274**, 47 (1992).

[28] A. E. Faraggi, "Yukawa couplings in superstring derived standard like models," Phys. Rev. **D47**, 5021 (1993).

[29] A. E. Faraggi, "Top quark mass prediction in superstring derived standard - like model," Phys. Lett. **B377**, 43 (1996).

[30] A. E. Faraggi, "Calculating fermion masses in superstring derived standard - like models," Nucl. Phys. **B487**, 55 (1997).

[31] G. B. Cleaver, "Advances in old-fashioned heterotic string model building," Nucl. Phys. Proc. Suppl. **62**, 161 (1998).

[32] G. B. Cleaver and A. E. Faraggi, "On the anomalous U(1) in free fermionic superstring models," Int. J. Mod. Phys. **A14**, 2335 (1999).

[33] G. Cleaver, M. Cvetic, J. R. Espinosa, L. L. Everett, and P. Langacker, "Classification of flat directions in perturbative heterotic superstring vacua with anomalous U(1)," Nucl. Phys. **B525**, 3 (1998).

[34] G. Cleaver, M. Cvetic, J. R. Espinosa, L. L. Everett, and P. Langacker, "Flat directions in three-generation free-fermionic string models," Nucl. Phys. **B545**, 47 (1999).

[35] G. Cleaver et al., "Physics implications of flat directions in free fermionic superstring models. I: Mass spectrum and couplings," Phys. Rev. **D59**, 055005 (1999).

[36] G. Cleaver et al., "Physics implications of flat directions in free fermionic superstring models. II: Renormalization group analysis," Phys. Rev. **D59**, 115003 (1999).

[37] G. B. Cleaver, "Quark masses and flat directions in string models," (1998).

[38] G. B. Cleaver, A. E. Faraggi, and D. V. Nanopoulos, "String derived MSSM and M-theory Unification," Phys. Lett. **B455**, 135 (1999).

[39] G. B. Cleaver, A. E. Faraggi, and D. V. Nanopoulos, "A minimal superstring standard model. I: Flat directions," Int. J. Mod. Phys. **A16**, 425 (2001).

[40] G. B. Cleaver, A. E. Faraggi, D. V. Nanopoulos, and J. W. Walker, "Phenomenological study of a minimal superstring standard model," Nucl. Phys. **B593**, 471 (2001).

[41] G. B. Cleaver, "M-fluences on string model building," (1999).

[42] G. B. Cleaver, A. E. Faraggi, D. V. Nanopoulos, and J. W. Walker, "Non-Abelian flat directions in a minimal superstring standard model," Mod. Phys. Lett. **A15**, 1191 (2000).

[43] G. B. Cleaver, A. E. Faraggi, and C. Savage, "Left-right symmetric heterotic-string derived models," Phys. Rev. **D63**, 066001 (2001).

[44] G. B. Cleaver, A. E. Faraggi, D. V. Nanopoulos, and J. W. Walker, "Phenomenology of non-Abelian flat directions in a minimal superstring standard model," Nucl. Phys. **B620**, 259 (2002).

[45] G. B. Cleaver, D. J. Clements, and A. E. Faraggi, "Flat directions in left-right symmetric string derived models," Phys. Rev. **D65**, 106003 (2002).

[46] G. B. Cleaver, A. E. Faraggi, and S. Nooij, "NAHE-based string models with SU(4) x SU(2) x U(1) SO(10) subgroup," Nucl. Phys. **B672**, 64 (2003).

[47] G. B. Cleaver, "Parameter space investigations of free fermionic heterotic models," (2002).

[48] G. Cleaver et al., "On the possibility of optical unification in heterotic strings," Phys. Rev. **D67**, 026009 (2003).

[49] J. Perkins et al., "Heterotic string optical unification," (2003).

[50] J. Perkins et al., "Stringent Phenomenological Investigation into Heterotic String Optical Unification," Phys. Rev. **D75**, 026007 (2007).

[51] G. B. Cleaver, A. E. Faraggi, E. Manno, and C. Timirgaziu, "Quasi-realistic heterotic-string models with vanishing one-loop cosmological constant and perturbatively broken supersymmetry?," Phys. Rev. **D78**, 046009 (2008).

[52] J. Greenwald et al., "Note on a NAHE Variation," (2009).

[53] G. Cleaver et al., "Investigation of Quasi–Realistic Heterotic String Models with Reduced Higgs Spectrum," (2011).

[54] G. B. Cleaver, "Supersymmetries in free fermionic strings," Nucl. Phys. **B456**, 219 (1995).

[55] H. Kawai, D. C. Lewellen, and S. H. H. Tye, "Classification of Closed Fermionic String Models," Phys. Rev. **D34**, 3794 (1986).

[56] H. Kawai, D. C. Lewellen, J. A. Schwartz, and S. H. H. Tye, "The Spin Structure Construction of String Models and Multiloop Modular Invariance," Nucl. Phys. **B299**, 431 (1988).

[57] D. C. Lewellen, "Embedding Higher Level Kac-Moody Algebras in Heterotic String Models," Nucl. Phys. **B337**, 61 (1990).

[58] S. Chaudhuri, S. W. Chung, G. Hockney, and J. D. Lykken, "String consistency for unified model building," Nucl. Phys. **B456**, 89 (1995).

[59] G. Aldazabal, A. Font, L. E. Ibanez, and A. M. Uranga, "String GUTs," Nucl. Phys. **B452**, 3 (1995).

[60] G. B. Cleaver, "What's new in stringy SO(10) SUSY GUTs," (1995).

[61] G. B. Cleaver, "Grand unified theories from superstrings," (1996).

[62] Z. Kakushadze and S. H. H. Tye, "A classification of three-family SO(10) and E(6) grand unification in string theory," Phys. Rev. **D55**, 7878 (1997).

[63] J. Erler, "Asymmetric orbifolds and higher level models," Nucl. Phys. **B475**, 597 (1996).

[64] Z. Kakushadze, G. Shiu, S. H. H. Tye, and Y. Vtorov-Karevsky, "A review of three-family grand unified string models," Int. J. Mod. Phys. **A13**, 2551 (1998).

[65] I. Antoniadis, J. R. Ellis, J. S. Hagelin, and D. V. Nanopoulos, "The Flipped SU(5) x U(1) String Model Revamped," Phys. Lett. **B231**, 65 (1989).

[66] R. N. Mohapatra and V. L. Teplitz, "Structures in the mirror universe," Astrophys. J. **478**, 29 (1997).

[67] R. N. Mohapatra and V. L. Teplitz, "Mirror matter MACHOs," Phys. Lett. **B462**, 302 (1999).

[68] R. N. Mohapatra and V. L. Teplitz, "Mirror dark matter," (2000).

[69] R. N. Mohapatra, S. Nussinov, and V. L. Teplitz, "Mirror matter as self interacting dark matter," Phys. Rev. **D66**, 063002 (2002).