Università degli Studi di Padova

Department of Physics and Astronomy "Galileo Galilei"
Master's Degree course in Theoretical Physics of Fundamental Interactions

Machine learning supergravity vacua

Thesis supervisor:
Prof. Dall'Agata Gianguido
Candidate:
Costantini Riccardo

Academic Year 2023-2024

Abstract

In this thesis we study maximally symmetric vacua for maximal supergravity in 4D. We will begin by showing that the structure of those vacua is determined by the gauging procedure. The consistency of the gauging, together with the extremization of the scalar potential, allows us to parameterize a system of equations that describes the vacua structure.
We also examine various approaches from the literature that utilize machine learning techniques to solve these equations. In particular, we present a novel approach based on a neural network architecture that improves the efficiency of those machine learning methods.
Finally, we will present an efficient algorithm for analytically solving systems of polynomial equations. This algorithm has been improved and implemented in a Python library, PyXLTensor. This library facilitates the writing, manipulation and solving of tensor expressions. Given its versatility it offers wide applicability in other fields where tensor equations need to be solved.

Contents

1 Introduction **5**

2 Maximal supergravity and its vacua **6**

2.1 Maximal supergravity in 4D 6

2.2 Gaugings and the scalar potential 9

2.3 "Going to the origin" and vacua 12

2.4 Known solutions 13

3 Machine learning supergravity vacua **14**

3.1 Learning machine learning 14

3.2 Old attempts 18

3.3 New attempts 20

3.3.1 Cluster algorithm 20

3.3.2 NS algorithm 22

4 Relinearization and new algorithms **26**

4.1 Algebraic approaches 26

4.1.1 Relinearization algorithm 26

4.1.2 XL algorithm 27

4.2 Improvements on the XL algorithm 29

4.3 Examples 31

5 Summary and outlooks **33**

A The group $\mathbf{E}_{7(7)}$ **34**

B How to deal with matrices with 0 positive(negative) eigenvalue 35

C How to use PyXLTensor 36

C.1 Initialization of the tensors 36

C.2 Basic operations 37

C.3 Block tensors 38

C.4 Symmetrization, anti-symmetrization, duality, δ and ϵ 39

C.5 Managing the indices 40

C.6 Elements of the tensors 41

C.7 Reading the tensors 42

C.8 Initializing a system of equations 42

C.9 Obtaining and reading the solutions 43

1 Introduction

The Standard Model of particle physics has been very successful in describing the electromagnetic, weak and strong interactions among particles. However, incorporating gravitational interactions in a consistent manner presents a significant challenge. One way to accommodate gravity in this picture can be achieved with supersymmetry. This provides an elegant solution, as supergravity naturally emerges when transitioning from global to local supersymmetry, analogous to how the other force-carrying fields in the Standard Model arise from localizing their respective symmetries.
Supergravity can be realized in various forms and each of them has its peculiarity. Of particular interest are the theories with the maximum possible amount of supersymmetry, known as maximal supergravity. These theories are particularly interesting because the stringent constraints imposed by maximal supersymmetry fully determine the structure of the theory through its gauging.
However, extended supersymmetric theories are non-chiral and therefore incompatible with a chiral theory such as the Standard Model. Despite this, maximal supergravity theories remain of physical interest due to the structure of their vacua and the emergence of residual symmetries. Concepts from these theories could provide crucial insights into open problems such as supersymmetry breaking, the dynamical selection of the vacuum state, the cosmological constant problem and the mechanisms for cosmic inflation.

To this day, there is no complete classification of vacua for maximal supergravity in four dimensions. This is due to the large number of degrees of freedom involved in the gauging procedure and the fact that the scalar potential of the theory depends on numerous scalar fields. Over the years, various techniques have been developed to address this challenge. These include theoretical developments, such as restricting the potential to specific families of scalar fields or describing the vacuum solely in terms of the gauging, as well as more efficient algorithms for solving the resulting system of equations.

In this thesis, we present some algorithms designed to find vacua solutions in maximal supergravity. Two distinct techniques will be discussed: one new approach is based on machine learning, specifically utilizing neural networks, and the other is derived from traditional algorithms in cryptography, which appears to be the more promising of the two approaches. The latter technique has led to the development of an open-source Python library, PyXLTensor, which facilitates the writing and solving of polynomial equations using tensors.
This library was build with the goals of being user-friendly and capable of solving quickly polynomial equations involving multiple variables, making it a useful tool not only within the field of quantum gravity but also for a wide range of applications.

First, we will review the essential theoretical aspects of maximal supergravity, focusing on how the scalar potential arises from the gauging procedure and the relative constraints. Next, we will introduce the computational techniques based on machine learning, discussing existing approaches and a novel method proposed in this thesis. Finally we will discuss how system of polynomial equations can be effectively solved. We will argue that with adequate hardware, a complete classification of the gaugings of the groups SU(4) and SU(3) can be done.

2 Maximal supergravity and its vacua

In this chapter we review maximal supergravity in 4D focusing on the necessary details to show how the vacua are related with the gauging procedure. We analyze the structure of the scalar manifold and the embedding tensor and show how they enter in the lagrangian, in particular in the scalar potential of the theory. Furthermore we show how the minimization problem of the potential with respect to the scalar fields can be related to a set of quadratic conditions on the embedding tensor. In the end we report the known solution for maximally symmetric vacua for maximal supergravity in 4D.

2.1 Maximal supergravity in 4D

In 4 space-time dimensions the maximum number of supersymmetric generators allowed in order to not have particles with spin grater than 2 is $\mathcal{N} = 8$. The field theoretic counterpart of such multiplet contains:

- 1 spin-2 graviton: e^a_μ;
- 8 spin-3/2 gravitinos: ψ^i_μ;
- 28 spin-1 vector fields: A^{ij}_μ;
- 56 spin-1/2 matter fields: χ^{ijk};
- 70 spin-0 scalars: ϕ^{ijkl}.

Here the indices i, j, k, l belong to the fundamental representation of SU(8) (the R-symmetry group) and are totally antisymmetrized.
Since in this thesis we are interested in classifying maximally symmetric vacua, the only objects that can get a v.e.v. are the scalar fields and the metric and the latter will take the form corresponding to Anti-de Sitter, Minkowski or de Sitter spacetimes.
In extended supergravity the scalar potential can only be generated through a gauging procedure, so in order to have a non trivial vacuum we need to introduce local gauge symmetries [1]. Before discussing the gauging procedure it is important to know what groups G_g can be gauged. The symmetries of the lagrangian are closely related to the structure of the scalar manifold $\mathcal{M}_{scalar}$ so we will briefly review the property of this space.
For $\mathcal{N} \geq 3$ supersymmetries and dualities constrain $\mathcal{M}_{scalar}$ to be an homogeneous space, i.e. a space where all the points can be connected by the action of an element of the isometry group G. Calling H the subgroup of G whose action leaves a chosen point (x) of the manifold unchanged we obtain that a generic point (x') is invariant under a transformation isomorphic to H

$$x' = gx = gHx = gHg^{-1}\ x', \tag{1}$$

so $\mathcal{M}_{scalar}$ can be identified with the coset space G/H.
In $4D$ the Hodge dual of a vector field is still a vector field. This implies the existence of a generalized electric-magnetic duality for their system of Bianchi identities and equations of motion [2]. For $\mathcal{N} = 8$ supergravity this group of dualities is realized by $\mathrm{Sp}(56, \mathbb{R})$ transformations acting linearly on the vectors and their duals. When the vector fields are coupled to other matter fields the $\mathrm{Sp}(56, \mathbb{R})$ duality group gets reduced to a subgroup G. For $N = 8$ supergravity $G = \mathrm{E}_{7(7)}$, a non-compact version of the E_7 exceptional group, with 70 non compact generators and 63 compact generators. The 28 vector fields of the supergravity multiplet and their duals transform altogether in the fundamental representation of $\mathrm{E}_{7(7)}$, which is 56-dimensional.
The scalar fields of the supergravity multiplet are coordinates of the coset manifold obtained by the quotient of $G = \mathrm{E}_{7(7)}$ with its maximal compact subgroup $H = \mathrm{SU}(8)/\mathbb{Z}_2$, and therefore we have a correspondence between the 70 scalar fields and the 70 non-compact generators acting non-trivially on any point of the scalar manifold $\mathcal{M}_{scalar} = G/H$.
A detailed discussion of the structure of $\mathrm{E}_{7(7)}$ is given in the appendix. We will however give here some results that are useful to describe the structure of $\mathcal{M}_{scalar}$. We can use the basis of $\mathrm{E}_{7(7)}$ in which the compact and noncompact generators are evident to isolate the noncompact generators simply by identifying the scalar fields ϕ_{ijkl} with the coefficients σ_{ijkl} that define this basis.
In this way the coset representatives L can be identified with

$$L(\phi)_{\underline{M}}{}^{\underline{N}} = \exp \begin{pmatrix} 0 & \phi_{ijkl} \\ \phi^{ijkl} & 0 \end{pmatrix}. \tag{2}$$

Since vector fields are real, it is natural to use a basis where the action of the duality group is also real and therefore it is convenient to consider the real section of $\mathrm{SU}(8)_{\mathbb{C}}$ provided by $\mathrm{SL}(8, \mathbb{R})$, rather than $\mathrm{SU}(8)_{\mathbb{R}}$. For this reason we will often employ a change of basis by means of the matrix S interpolating between the two real forms. In detail, in order to relate the scalar and vector fields, the coset representatives can be written using the constant tensor $S_{\underline{M}}{}^{N}$, whose explicit form is given in the appendix,

$$L(\phi)_{M}{}^{\underline{N}} = S^{\dagger}_{M}{}^{\underline{P}} L(\phi)_{\underline{P}}{}^{\underline{N}}, \tag{3}$$

so that M is an index in the fundamental representation of $\mathrm{SL}(8, \mathbb{R})$ and $\underline{N}$ is an index in the fundamental representation of SU(8). The plain indices will be used to represent electric and magnetic indices in the $\mathrm{SL}(8, \mathbb{R})$ representation, for example $V^M = \left(V^\Lambda, V_\Lambda\right)$, with $\Lambda = 1, \ldots, 28$. The underline indices will be used to represent electric and magnetic indices in the SU(8) representation, for example $V^{\underline{M}} = \left(V^{ij}, V_{ij}\right)$, with $i, j = 1, \ldots, 8$ and ij are a couple of antisymmetric indices associated with the fundamental representation of the R-symmetry group SU(8).

This S tensor is used to pass between the explicit form of the generators of $\mathrm{E}_{7(7)}$ decomposed under SU(8) and SL(8). The eq. (3) allows for a more natural integration of the scalar fields in the lagrangian with respect to eq. (2). Since the vector fields will be defined using SL(8) covariant indices, this tensor allows us to relate the scalar manifold with the gauging through the T-tensor as we will see later.

For completeness we report the ungauged lagrangian, without giving its derivation, which can be found in [1].
It is useful to define the the field strengths as a sum of selfdual and antiselfdual fields as follows

$$F^{\Lambda}_{\mu\nu} = F^{+\ \Lambda}_{\mu\nu} + F^{-\ \Lambda}_{\mu\nu}, \qquad \tilde{F}^{\pm\ \Lambda}_{\mu\nu} = \pm F^{\pm\ \Lambda}_{\mu\nu}. \tag{4}$$

Here we will use Greek uppercase indices, $\Lambda, \Sigma = 1, \ldots, 28$, to represent electric vector fields, with an upper index, and magnetic vector fields, with a lower index, for example the coset representative in the SL(8) basis is

$$L_M{}^{\underline{N}} = \begin{pmatrix} L_{\Lambda}{}^{ij} & L_{\Lambda kl} \\ L^{\Sigma ij} & L^{\Sigma}{}_{kl} \end{pmatrix}. \tag{5}$$

Up to 4-fermi interaction, the lagrangian reads:

$$\begin{aligned} \mathcal{L} = & -\frac{1}{2}eR - \frac{1}{2}\varepsilon^{\mu\nu\rho\sigma}(\bar{\psi}_\mu{}^i\gamma_\nu D_\rho\psi_{\sigma i} - \bar{\psi}_\mu{}^i\overleftarrow{D}_\rho\gamma_\nu\psi_{\sigma i}) \\ & - \frac{i}{4}e\left(\mathcal{N}_{\Lambda\Sigma}F^{+\ \Lambda}_{\mu\nu}F^{+\mu\nu\Sigma} - \bar{\mathcal{N}}_{\Lambda\Sigma}F^{-\ \Lambda}_{\mu\nu}F^{-\mu\nu\Sigma}\right) \\ & - \frac{1}{12}e(\bar{\chi}^{ijk}\gamma^\mu D_\mu\chi_{ijk} - \bar{\chi}^{ijk}\overleftarrow{D}_\mu\gamma^\mu\chi_{ijk}) - \frac{1}{12}e\left|\mathcal{P}^{ijkl}_\mu\right|^2 \\ & - \frac{\sqrt{2}}{6}e\left(\bar{\chi}_{ijk}\gamma^\nu\gamma^\mu\psi_{\nu l}\mathcal{P}^{ijkl}_\mu + \text{h.c.}\right) \\ & + eF^{+\ \Lambda}_{\mu\nu}\mathcal{O}^{+\mu\nu}_\Lambda + eF^{-\ \Lambda}_{\mu\nu}\mathcal{O}^{-\mu\nu}_\Lambda. \end{aligned} \tag{6}$$

The first three lines contain the kinetic terms for the fields. The first line contains the kinetic term for the graviton and the gravitini; the second line contains the kinetic term for the vector fields, where the field-dependent tensor $\mathcal{N}_{\Lambda\Sigma}$ comprises the field dependent generalized theta angles and coupling constants; the third line contains the kinetic term for the matter fields and the scalar field. The gauge kinetic matrix $\mathcal{N}$ is determined by the equation

$$L^{\Sigma ij}\mathcal{N}_{\Sigma\Lambda} = -L_\Lambda{}^{ij}. \tag{7}$$

The $\mathcal{P}$ tensor is the vielbein on the scalar manifold

$$\begin{aligned} \mathcal{P}_{\mu\ ijkl} &= i\Omega^{MN}L_{Mij}D_\mu L_{Nkl} = i(L_{\Lambda ij}D_\mu L^\Lambda_{kl} - L^\Lambda_{ij}D_\mu L_{\Lambda kl}), \\ \mathcal{P}^{ijkl}_\mu &= \frac{1}{24}\varepsilon^{ijklmnqp}\mathcal{P}_{\mu\ mnpq}, \end{aligned} \tag{8}$$

where Ω is the symplectic invariant 56×56 matrix.
The fourth line describe couplings between scalars and fermions, with the fermionic bilinear $\mathcal{O}$ given by

$$\mathcal{O}^{+\ ij}_{\mu\nu} = \frac{1}{2}\sqrt{2}\bar{\psi}^i_\rho\gamma^{[\rho}\gamma_{\mu\nu}\gamma^{\sigma]}\psi^j_\sigma - \frac{1}{2}\bar{\psi}_{\rho k}\gamma_{\mu\nu}\gamma^\rho\chi^{ijk} - \frac{1}{144}\sqrt{2}\varepsilon^{ijklmnpq}\bar{\chi}_{klm}\gamma_{\mu\nu}\chi_{npq} \tag{9}$$

Finally the fifth line collects the Pauli-like terms.
In the above expressions all the derivatives D_μ are covariant with respect to diffeomorphisms, Lorentz and SU(8) transformations.
As expected, the lagrangian is invariant under a $\mathcal{N} = 8$ supersymmetric transformation

and the corresponding variations of the fields are:

$$\begin{aligned}
\delta\psi_\mu{}^i =& 2D_\mu\epsilon^i + \frac{\sqrt{2}}{4}\hat{F}^{-ij}_{\rho\sigma}\gamma^{\rho\sigma}\gamma_\mu\epsilon_j + \frac{1}{4}\bar{\chi}^{ikl}\gamma^a\chi_{jkl}\gamma_a\gamma_\mu\epsilon^j \\
&+ \frac{\sqrt{2}}{2}\bar{\psi}_{\mu k}\gamma^a\chi^{ijk}\gamma_a\epsilon_j - \frac{1}{576}\varepsilon^{ijklmnpq}\bar{\chi}_{klm}\gamma^{ab}\chi_{npq}\gamma_\mu\gamma_{ab}\epsilon_j, \\
\delta\chi^{ijk} =& -2\sqrt{2}\hat{\mathcal{P}}^{ijkl}_\mu\gamma^\mu\epsilon_l + \frac{3}{2}\hat{F}^{-[ij}_\mu\nu\gamma^{\mu\nu}\epsilon^{k]} - \frac{\sqrt{2}}{24}\varepsilon^{ijklmnpq}\bar{\chi}_{lmn}\chi_{pqr}\epsilon^r, \\
\delta e_\mu{}^a =& \bar{\epsilon}^i\gamma^a\psi_{\mu i} + \bar{\epsilon}_i\gamma^a\psi_\mu{}^i, \\
\delta L_M^{ij} =& 2\sqrt{2}L_{Mkl}\left(\bar{\epsilon}^{[i}\chi^{jkl]} + \frac{1}{24}\varepsilon^{ijklmnpq}\bar{\epsilon}_m\chi_{npq}\right), \\
\delta A_\mu{}^M =& -i\Omega^{MN}L_N^{ij}\left(\bar{\epsilon}^k\gamma_\mu\chi_{ijk} + 2\sqrt{2}\bar{\epsilon}_i\psi_{\mu j}\right) + \text{h.c.},
\end{aligned} \tag{10}$$

where the ϵ_i are the infinitesimal spinorial parameters of the supersymmetry transformation.

2.2 Gaugings and the scalar potential

In order to obtain a consistent gauging procedure, it is not enough to specify the gauge group $G_g \subset \mathrm{E}_{7(7)}$, but one also needs its symplectic embedding. This requirement follows from the fact that, already at the ungauged level, we can differentiate equivalent and non-equivalent lagrangians by means of symplectic transformations. In fact, while the Bianchi identities and the equations of motion of the vector fields are invariant under $\mathrm{Sp}(2n_v,\mathbb{R})$, where n_v is the number of vector fields in the lagrangian, the rest of the equations of motion are only invariant under a smaller group $G \subset \mathrm{Sp}(2n_v,\mathbb{R})$ ($G = \mathrm{E}_{7(7)}$ for $\mathcal{N}=8$, where $n_v = 28$). We will therefore obtain different theories if we act with $\mathrm{Sp}(2n_v,\mathbb{R})$ transformations that are not contained in G and hence we will obtain different gauge models is such theories allow for the gauging of the same G_g.
Moreover, once a lagrangian is fixed, one can always use local fields redefinition of the n_v vectors in the lagrangian given by $\mathrm{SL}(n_v,\mathbb{R})$ transformations. Altogether, this implies that the set of inequivalent lagrangians is identified with the quotient

$$\mathrm{GL}(28,\mathbb{R})\backslash\mathrm{Sp}(56)/\mathrm{E}_{7(7)}. \tag{11}$$

In order to keep track of all the possible inequivalent theories a useful technique is the embedding tensor formalism [1,3], which we now briefly review.
The local symmetries of the ungauged theory are the abelian transformations $U(1)^{28}$ of the vector fields. In order to chose which vectors will be gauged using the generators $t_\alpha \in \mathfrak{e}_{(7)}$ we will define the gauge generators as

$$X_M = \Theta_M{}^\alpha t_\alpha \tag{12}$$

and use them to write the covariant derivatives

$$\mathcal{D}_\mu = \partial_\mu - A^M_\mu X_M = \partial_\mu - A^M_\mu\Theta_M{}^\alpha t_\alpha. \tag{13}$$

The embedding tensor Θ is therefore, in our case, a 56×133 matrix with rank equal to the dimension of the gauge group. This tensor allow us to define every possible gauging without needing it to be specified at priori. However the embedding tensor can not be chosen arbitrarily and must respect some constraints.

The first constraint follows from the requirement that the action of the gauge symmetry must not transform the embedding tensor because we want the lagrangian to remain invariant under the action of G_g, hence,

$$0 = \delta_M \Theta_N{}^\alpha = \Theta_M{}^\beta \delta_\beta \Theta_N{}^\alpha = \Theta_M{}^\beta [t_\beta]_N{}^P \Theta_P{}^\alpha + \Theta_M{}^\beta [t_\beta^{\text{adj.}}]_\gamma{}^\alpha \Theta_N{}^\gamma. \tag{14}$$

Contracting the above expression with the generator t_α, the result is

$$\Theta_M{}^\beta \left([t_\beta^{\text{adj.}}]_\gamma{}^\alpha t_\alpha \right) \Theta_N{}^\gamma = - \left(\Theta_M{}^\beta t_{\beta N}{}^P \right) \Theta_P{}^\alpha t_\alpha. \tag{15}$$

Defining $X_{MN}{}^P = \Theta_M{}^\beta t_{\beta N}{}^P$, the above expression can be rewritten as

$$[X_M, X_N] = -X_{MN}{}^P X_P \tag{16}$$

which resembles the standard closure of the gauge algebra, though we notice that while $X_{MN}{}^P$ is like a structure constant for the gauging, $X_{(MN)}{}^P$ can be non zero.

The second constraint is necessary in order to have 28 vector fields that are mutually local. This implies that the embedding tensor should satisfy

$$\Theta_M{}^\alpha \Theta_N{}^\beta \Omega^{MN} = 0. \tag{17}$$

This guarantees that there exists a field redefinition such that the gauging is done using only electric vector fields.

Finally we have a supersymmetry constraint. The embedding tensor a priori belongs to the $\mathrm{E}_{7(7)}$ representations

$$\mathbf{56} \otimes \mathbf{133} = \mathbf{56} \oplus \mathbf{912} \oplus \mathbf{6480}, \tag{18}$$

but only the **912** representation is compatible with supersymmetry. As we will see shortly, this is related to the supersymmetric variations of the fermionic fields after the gauging procedure. Such variations are modified to ensure that supersymmetric invariance is preserved. Calling $\mathbb{P}_{(912)}$ the projector operator on the **912** representation, we need to impose $\mathbb{P}_{(912)}\Theta = \Theta$. This constraint can be written as [2]:

$$\begin{aligned} t_{\alpha M}{}^N \Theta_N{}^\alpha &= 0, \\ (t_\beta t^\alpha)_M{}^N \Theta_N{}^\beta &= -\frac{1}{2} \Theta_M{}^\alpha. \end{aligned} \tag{19}$$

Here the indices are raised and lowered through the Cartan metric $\eta_{\alpha\beta} = \mathrm{Tr}\left(t_\alpha^{\text{adj.}} t_\beta^{\text{adj.}}\right)$.

In the lagrangian the embedding tensor will appear as a modification of the variation of the fermionic fields under a supersymmetric transformation and through contractions of the so called T-tensor,

$$T_{\underline{MN}}{}^{\underline{P}}[\Theta, \phi] = L_{\underline{M}}^{-1Q} L_{\underline{N}}^{-1R} X_{QR}{}^S L_S{}^{\underline{P}}. \tag{20}$$

When gauging a supergravity theory, the introduction of covariant derivatives of order $\mathcal{O}(g)$ leads to the introduction of new terms of order $\mathcal{O}(g)$ and $\mathcal{O}(g^2)$ to the ungauged lagrangian. In order to restore supersymmetry the transformation rule for the fermions must be modified adding a term of order $\mathcal{O}(g)$, the so called "fermionic shift":

$$\begin{aligned}\delta'\psi^i_\mu &= \sqrt{2}gA_1^{ij}\gamma_\mu\epsilon_j,\\ \delta'\chi^{ijk} &= -2gA_{2l}{}^{ijk}\epsilon^l.\end{aligned} \tag{21}$$

With $A_1^{ij} = A_1^{(ij)}$, $A_{2i}{}^{jkl} = A_{2i}{}^{\{jkl\}}$, $A_{2i}{}^{ijk} = 0$. Those new terms must be added to the respective transformation rules in eq. (10) and the derivative in $\delta\psi$ must be substituted with the covariant derivative eq. (13). The tensors A_1 and A_2 uniquely determine the T-tensor, in fact we can notice that under SU(8) the **912** representation of $\mathbf{56}\otimes\mathbf{133}$ is broken into

$$\mathbf{36}\oplus\overline{\mathbf{36}}\oplus\mathbf{420}\oplus\overline{\mathbf{420}} \tag{22}$$

which precisely match the 36 and 420 complex degrees of freedom of A_1 and A_2. Explicitly:

$$T_{\underline{MN}}{}^{\underline{P}} = \left(T_{ij\underline{N}}{}^{\underline{P}},\ T^{kl}{}_{\underline{N}}{}^{\underline{P}}\right), \tag{23}$$

$$\begin{aligned}T_{ij} &= \begin{pmatrix} -\frac{2}{3}\delta_{[k}{}^{[p}T^{q]}{}_{l]ij} & \frac{1}{24}\epsilon_{klrstuvw}T^{tuvw}{}_{ij}\\ T^{mnpq}{}_{ij} & \frac{2}{3}\delta_{[r}{}^{[m}T^{n]}{}_{s]ij}\end{pmatrix},\\ T^{ij} &= \begin{pmatrix} \frac{2}{3}\delta_{[k}{}^{[p}T_{l]}{}^{q]ij} & T_{klrs}{}^{ij}\\ \frac{1}{24}\epsilon^{mnpqtuvw}T_{tuvw}{}^{ij} & -\frac{2}{3}\delta_{[r}{}^{[m}T_{s]}{}^{n]ij}\end{pmatrix},\end{aligned} \tag{24}$$

with $T_k{}^{lij} = -\frac{3}{4}A_{2k}{}^{lij} - \frac{3}{2}A_1{}^{l[i}\delta^{j]}{}_k$ and $T_{klmn}{}^{ij} = -\frac{4}{3}\delta^{[i}{}_{[k}T^{j]}{}_{lmn]}$.

The modification of the fermionic supersymmetric transformation forces us to introduce a Yukawa-like term in the lagrangian,

$$\mathcal{L}_{\rm Y} = eg\left(\frac{\sqrt{2}}{2}A_{1ij}\overline{\psi}_\mu{}^i\gamma^{\mu\nu}\psi_\nu{}^j + \frac{1}{6}A_{2i}{}^{jkl}\overline{\psi}_\mu{}^i\gamma^\mu\chi_{jkl} + \frac{\sqrt{2}}{144}\epsilon^{ijkpqr[lm}A_2{}^{n]}{}_{pqr}\overline{\chi}_{ijk}\chi_{lmn}\right) + \text{h.c.}\ . \tag{25}$$

Substituting the vacuum expectation value for the scalar fields in this expression we obtain the mass terms for the fermionic fields.
In order to cancel the variations of this new term a scalar potential of order $\mathcal{O}(g^2)$ must be added,

$$\begin{aligned}V &= g^2\left(\frac{1}{24}A_{2i}{}^{jkl}A_2{}^i{}_{jkl} - \frac{3}{4}A_1^{ij}A_{1il}\right),\\ V &= \frac{g^2}{672}\left(X_{MN}{}^RX_{PQ}{}^S\mathcal{M}^{MP}\mathcal{M}^{NQ}\mathcal{M}_{RS} + 7X_{MN}{}^QX_{PQ}{}^N\mathcal{M}^MP\right),\end{aligned} \tag{26}$$

where $\mathcal{M}_{MN} = L_M{}^{\underline{P}}L_{N\underline{P}}$.

Once this is done the theory is consistent and no other terms of order $\mathcal{O}(g^3)$ are needed to restore supersymmetry.

2.3 "Going to the origin" and vacua

In order to find the possible maximally symmetric vacua one needs to evaluate the scalar fields for which V is a critical points. This however is a nontrivial task. Another possibility to find all possible vacua lies in the properties of $\mathcal{M}_{scalar}$. Since any two point on this manifold can be mapped into each other via the action of a $\mathrm{E}_{7(7)}$ element, we can map any point to $\phi = 0$. Crucially, the scalar potential is invariant under this transformation. Doing so we can focus on a fixed point of the scalar manifold and consider variations of the potential with respect to the embedding tensor, as we will now show. This analysis and results are discussed in detail in [3].
The representative $L(\phi_U)$ of a point ϕ_U of $\mathrm{E}_{7(7)}/\mathrm{SU}(8)$ is related to the representative $L(0)$ of the point $\phi = 0$ by

$$L(\phi_U)_{\underline{M}}{}^{N} = h(U)_{\underline{M}}{}^{\underline{P}} L(0)_{\underline{P}}{}^{Q} U_{Q}{}^{N}, \quad U \in \mathrm{E}_{7(7)},\ h \in \mathrm{SU}(8), \tag{27}$$

this is always possible because the scalar manifold is an homogeneous space. The same kind of shift can be done for the T-tensor associated with those scalar fields and, up to SU(8) transformations, is

$$\begin{aligned} T_{\underline{MN}}{}^{\underline{P}}[\phi_U] &= (L[0]U)^{-1Q}_{\underline{M}} (L[0]U)^{-1R}_{\underline{N}} X_{QR}{}^{S} (L[0]U)_{S}{}^{\underline{P}} \\ &= L[0]^{-1Q}_{\underline{M}} L[0]^{-1R}_{\underline{N}} X'_{QR}{}^{S} L[0]_{S}{}^{\underline{P}} = T'_{\underline{MN}}{}^{\underline{P}}[0], \end{aligned} \tag{28}$$

where $X'_{QR}{}^{S} = U_{Q}^{-1M} U_{R}^{-1N} X_{MN}{}^{P} U_{P}{}^{S}$ and we can ignore the SU(8) transformations of the indices of the T-tensor since this is a symmetry of our lagrangian. In this way we can study all the possible vacua by focusing on the origin of $\mathcal{M}_{scalar}$. Finding the minima of the potential for $\phi = 0$ is much easier than solving the minimization for ϕ because, as we will see shortly, those condition will be quadratic in terms of the embedding tensor. This allow us to study the vacua not only for a fixed embedding but for a family of them in one go.

The derivative of the coset representative for $\phi = 0$ can be simply evaluated using its definition eq. (2)

$$\partial_{ijkl} L(\phi)_{\underline{M}}{}^{\underline{N}} \Big|_{\phi=0} = L(0)_{\underline{M}}{}^{\underline{P}} [t_\rho]_{\underline{P}}{}^{\underline{N}}, \tag{29}$$

where instead of the indices $ijkl$ the ρ index is used to represent the non-compact generator of $\mathrm{E}_{7(7)}$. If we now consider a contraction between a coset representative and the embedding tensor, the action of the derivative of the scalar field can be seen as the variation of the embedding tensor, explicitly:

$$\partial_{ijkl} L(\phi)_{\underline{M}}{}^{\underline{N}} \Theta_{\underline{N}}{}^{\alpha} \Big|_{\phi=0} = L(0)_{\underline{M}}{}^{\underline{P}} \left([t_\rho]_{\underline{P}}{}^{\underline{N}} \Theta_{\underline{N}}{}^{\alpha} \right) \propto L(0)_{\underline{M}}{}^{\underline{P}} \delta_\rho \Theta_{\underline{N}}{}^{\alpha}. \tag{30}$$

This means that the derivative of the potential at the origin of the scalar manifold can be evaluated by using the variation of the embedding tensor. Furthermore, taking $\phi = 0$

makes $\mathcal{M} = \mathbb{1}$. Combining all of the above results we obtain

$$\begin{aligned} V(0,\Theta) &= \Theta_M{}^\alpha \Theta^{M\beta}(\delta_{\alpha\beta} + 7\eta_{\alpha\beta}), \\ \partial_\rho V(\phi,\Theta)\Big|_{\phi=0} &\propto \frac{\delta V}{\delta \Theta}\left[[t_\rho]_N{}^P \Theta_P{}^\alpha + [t_\rho^{\text{adj.}}]_\gamma{}^\alpha \Theta_N{}^\gamma\right] \\ &\propto \Theta_M{}^\alpha [t_\rho]^{MN} \Theta_N{}^\beta (\delta_{\alpha\beta} + 7\eta_{\alpha\beta}) + \Theta_M{}^\alpha \Theta^{M\beta} [t_\rho^{\text{adj.}}]_{\alpha\beta}, \\ M_\rho^\chi = \partial_\rho \partial^\chi V(\phi,\Theta)\Big|_{\phi=0} &\propto \Theta_M{}^\alpha [t_\rho t^\chi]^{MN} \Theta_N{}^\beta (\delta_{\alpha\beta} + 7\eta_{\alpha\beta}) + \Theta_M{}^\alpha \Theta^{M\beta} [t_\rho^{\text{adj.}} t^\chi_{\text{adj.}}]_{\beta\alpha} \\ &\quad + \Theta_M{}^\alpha [t_\rho]^{MN} \Theta_N{}^\beta [t^\chi_{\text{adj.}}]_{\beta\alpha} + \Theta_M{}^\alpha [t^\chi]^{MN} \Theta_N{}^\beta [t_\rho^{\text{adj.}}]_{\beta\alpha}. \end{aligned} \tag{31}$$

The mass matrix M_ρ^χ is useful in order to discuss perturbative stability of the vacua.

2.4 Known solutions

The search for vacua before the introduction of the embedding tensor and the idea of shifting the scalar fields at $\phi = 0$ has been very slow. Solution were found analytically or numerically for particular cases [4–19]. In particular, the first progress towards a better classification of multiple vacua was made in [19]. In this paper, machine learning techniques were used for the first time, making it possible to find multiple vacua at once. In [3] was introduced the shift of the scalar fields at $\phi = 0$ and this allowed to more easily search for solutions. With this technique a plethora of new (and old) vacua have been found [20–35].

The known vacua that have a residual non abelian gauge group can be found in

- De Sitter:
  [3, 21, 25]
- Minkowski:
  [3, 23–25]
- Anti-de Sitter:
  [3–5, 7, 13–15, 18–25, 27, 28, 30, 31, 33–35]

3 Machine learning supergravity vacua

In this chapter we will introduce machine learning and discuss how it can be used to find vacua in maximal supergravity. We begin by explaining the fundamental concepts of ML, using concrete examples to clarify these ideas. Next, we review existing implementations of ML in the context of supergravity vacua. Finally, we present a novel approach that leverages ML techniques to improve the search for supergravity vacua.

3.1 Learning machine learning

Machine learning (ML) is a subset of artificial intelligence where algorithms learn directly from data without explicit programming. Initially applied to fields like text, speech, and image recognition, ML has recently seen increased use in physics, from track reconstruction [36] to identifying phase transitions in many-body systems [37] and generating gauge-field configurations on the lattice [38].

The strength and versatility of ML stems from a very simple idea: the parameters of the model are not fixed at priori but are adjusted during iterations based on the seen data. This enables the model to optimize for specific tasks by searching the best parameters for it.
This can generally be done through a definition of a loss function and an update procedure. The loss function quantifies how good the model performed on a single data, the lower the value of the loss function, the better. The update procedure changes the free parameters of the model such that, given a single data (or a set of data), the model performs slightly better on that single data (or a set of data) than it did before the update. The loss function is used to perform this update. Given a data point, we can evaluate the loss function for that input, keeping the dependence of the result on the model parameters explicit. We can change the parameters slightly in such a way that the value of the loss function decreases. Generally this can be achieved with a small step in the opposite direction of the gradient in the parameter space of the loss function.
Concretely, let's consider the following example. Take $y = f(x)$, a multivariable function and a dataset D containing pairs of variables (x, y). If the function $f(x)$ is parametrized by a set of values $P = \{p_i\}$, we aim to fit $f(x)$ to the data by minimizing the loss function, often defined as the mean squared error:

$$\mathcal{L} = \sum_{k=0}^{N_{data}} \big|\big|y_k - f(x_k)\big|\big|^2. \tag{32}$$

Minimizing the loss means that the distance between the data and the interpolating function is as low as possible. In order to perform this minimization we can first evaluate the gradient, since the gradient in the parameter space points in the steepest direction of the function $\mathcal{L}[P]$. We can then lower the value of the loss with the update

$$p_i \to p_i - \eta \partial_i \mathcal{L}, \tag{33}$$

where ∂_i is the derivative with respect to p_i, and η is a small, fixed constant (often $\eta = 0.01$). The above update makes sure that, if η is nor to small nor to big, the loss

function decreases. Proper initialization of parameters allows convergence towards the global minimum of $\mathcal{L}$, yielding the best fit.
This method, known as gradient descent [39], is one of the many different algorithms used in ML.

While the optimization technique used in the above example works in theory, in practice it may be slow or fail to converge. To address these issues, various techniques have been developed over the years.
A crucial factor is the learning rate η, which must be carefully chosen. If η is too large, the parameters may overshoot the minimum or diverge. If too small, convergence may be excessively slow, as illustrated in fig. 1.

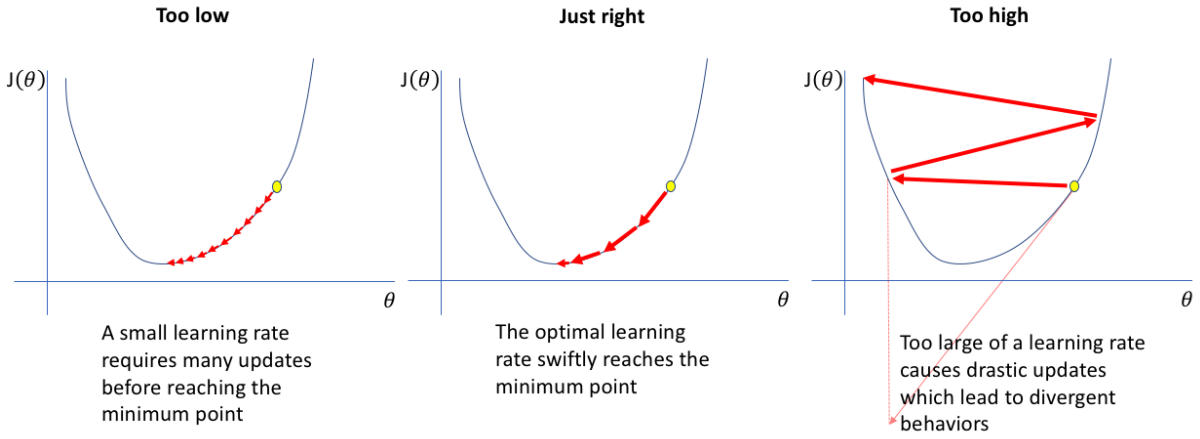

Figure 1: Effect of different learning rates: Left - learning rate too small, slow convergence; Center - optimal learning rate; Right - learning rate too large, no convergence. Figure taken from [40].

Beyond choosing a good initial learning rate, it is often advantageous to adjust η dynamically during training [41]. This can be achieved by decaying the learning rate over iterations or adapting it based on the loss function's value. The variation of the learning rate over the iterations can be implemented in various ways, for example the learning rate can be proportional to the inverse of the number of iteration or in each iteration it can be multiplied by a fixed constant smaller than 1. In the former the model takes larger steps at the beginning and gradually finer steps going on. For the latter the learning rate decreases at an exponential rate, this can guarantee quick convergence but the model can potentially converge before reaching the minimum.

Techniques like momentum, which incorporates past updates into the current step, can further accelerate convergence and avoid local minima by enabling the model to "escape" shallow basins in the loss landscape [42]. Momentum-enhanced gradient descent modifies the update rule:

$$w_i^{(n)} \to w_i^{(n+1)} = w_i^{(n)} - \Delta w_i^{(n)}, \tag{34}$$

where the superscript of the parameter w_i represent the number of the iteration. Without momentum we have

$$\Delta w_i^{(n)} = \eta \partial_i \mathcal{L}, \tag{35}$$

while with momentum the expression becomes

$$\Delta w_i^{(n)} = \eta \partial_i \mathcal{L} + \gamma \Delta w_i^{(n-1)}, \tag{36}$$

where γ is a constant between 0 and 1, allowing the algorithm to retain a fraction of the previous update's direction, which can smooth the path towards the global minimum.

Based on the ideas above, a lot of optimization strategies came over the years that involve adjusting simultaneously the learning rate and momentum. Some popular examples are:

- AdaGrad (Adaptive Gradient) [43]
  The learning rate is different at each iteration for each variable. The learning rate at the n^{th} iteration for the i^{th} parameter is

$$\eta_i^{(n)} = \frac{\eta}{\sqrt{\sum_{m=1}^{n} \left(\Delta w_i^{(m)}\right)^2}}. \tag{37}$$

  This allows the model to quickly escape saddle points.

- RMSprop (Root Mean-Square propagation) [44]
  Analogous to AdaGrad but with the difference that the sum of the square of the gradients include a exponential decay factor. This allows the model to retain information about only a few previous steps rather than the entire path. This also helps us avoid the potential problem of a vanishing learning rate.

- Adam (Adaptive moment estimation) [45]
  Similar to RMSprop but using momentum too. However, some corrections need to be made in the implementation, since this is a biased estimator, which we will not discuss here. This update procedure generally is the best performing one, it allows the model to quickly navigate the common saddle points of the higher dimensional loss function and it gives the possibility to escape its local minima.

In this thesis we attempted a novel approach to finding supergravity vacua by using ML techniques. Since this approach involves deep learning and particularly neural networks (NNs) [46], we will now introduce what a NN is with all the necessary concepts.
The structure of a NN takes inspiration by biological brains, hence the name. The building block of a NN is the neuron. Each neuron is connected with edges, like the synapses in a brain. The neurons are organised in layers. Each neuron receives inputs from neurons in the previous layer, processes these inputs, and transmits outputs to neurons in the next layer, as shown in the figure below.

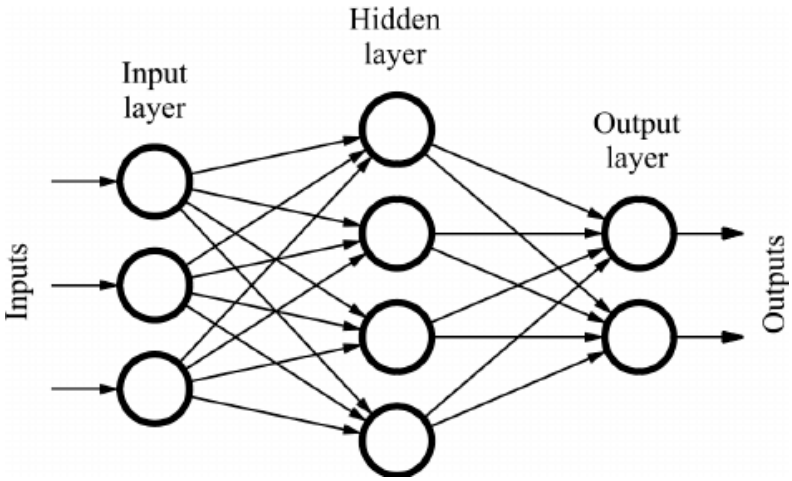

Figure 2: Structure of a feed-forward neural network (FFNN) with input, hidden, and output layers. The FFNN in this picture has three inputs, each neuron in the hidden layer has as an input from each neuron in the input layer. Similarly the outputs are obtained from the hidden layer. Figure taken from [47]

The output y_i of the i^{th} neuron in a layer is obtained from the outputs $\{x_j\}_{i=j,\dots,d}$ of the previous layer by

$$y_i = f(a_i), \quad a_i = b_i + \sum_{j=0}^{d} w_{ij} x_j, \tag{38}$$

where f is a non-linear function, called activation function and w_{ij} and b_i are a set of parameters called respectively weight and biases. The choice of activation functions is not a trivial task, a good activation function in general must balance various properties, such as ease of evaluation, along with its derivative, the absence of flat regions or the output range. A detailed list of various activation functions can be found in [48]. Generally the two most common activation function are ReLUs $\big(f(x) = (x + |x|)/2\big)$ and sigmoids $\big(f(x) = 1/(1 + e^{-x})\big)$.
A FFNN is essentially a sequence of layers, each consisting of an affine transformation followed by an activation function. There are no universal guidelines for determining the optimal structure of a neural network, selecting activation functions, or choosing the appropriate loss function. These decisions are typically made through a process of trial and error, with intuition and expertise developing over time.
The free parameters that the model must learn include the weights and biases at each layer, as well as any parameters associated with the activation functions, if present. Once the structure of a FFNN is defined, the next step is to select an appropriate loss function for the specific task. The training process then proceeds similarly to the methods discussed in section 3.1. The phase of evaluating the derivative of the loss with respect to the parameters is known in the context of neural networks as backpropagation. During the "forward" evaluation, intermediate outputs are temporarily stored, which facilitates efficient computation of derivatives through automatic differentiation [49]. Furthermore, the use of affine transformations and differentiable activation functions ensures rapid evaluation of these derivatives.

Despite its simple structure, a neural network is a powerful tool due to its ability to

model complex, non-linear relationships in data. As demonstrated in [50], FFNNs are universal approximators, meaning they can approximate any Borel measurable function to any desired degree of accuracy, provided that a sufficient number of hidden neurons are available.

3.2 Old attempts

Numerical algorithms used to find vacua of maximal supergravity are generally based on optimization techniques, often variants of gradient descent algorithms. Extensive search for solutions using stochastic gradient descent (SGD) was performed in [19, 31, 35, 51, 52]. In those paper, several sectors of extended supergravity in 4D or 5D were analyzed. The problem was approached in two different ways. Before [3], for a choice of the desired gauge group, a possible embedding of this group in $\mathrm{E}_{7(7)}$ was chosen. However the potential for this theory still depend on 70 scalar fields and therefore a complete search for the minima was not possible. The idea was to evaluate the potential on a submanifold of M_{scalar} such that the critical points on the submanifold are critical points on M_{scalar} too. The other technique was to search for minima using eq. (31) and parameterizing the embedding tensor so that the total number of degrees of freedom were manageable.
Either way the problem were reduced to find the critical points of the potential, given a certain parametrization. Since the critical points, by definition, have vanishing gradient, the technique used is a minimization of the loss function

$$\mathcal{L} = \left|\left|\partial V(x)\right|\right|^2, \tag{39}$$

with x the parameter that belong to the respective space, based on the approach chosen. The point is a critical point of the manifold only if, with a reasonable precision, $\mathcal{L} = 0$.

A more recent approach was tested in [53, 54], analyzing maximally symmetric vacua in 5D and 7D supergravity, employing the shift of the scalar field as in section 2.3, using both SGD and genetic algorithms [55]. The most notable technique was the Covariance Matrix Adaptation - Evolutionary Strategy [56] (CMA-ES). For this algorithm an initial population (in the case of the papers, set of points that parameterize a particular sector of the embedding tensor) is sampled at random from a multivariate normal distribution. Each generation the mean and covariant matrix of the population is updated following a maximum-likelihood principle for the minimization of the loss function (in the case of the papers, the scalar potential) and then the population is resampled. With careful initialization the algorithm can find global minimum of the potential. An example of the process is given in fig. 3.

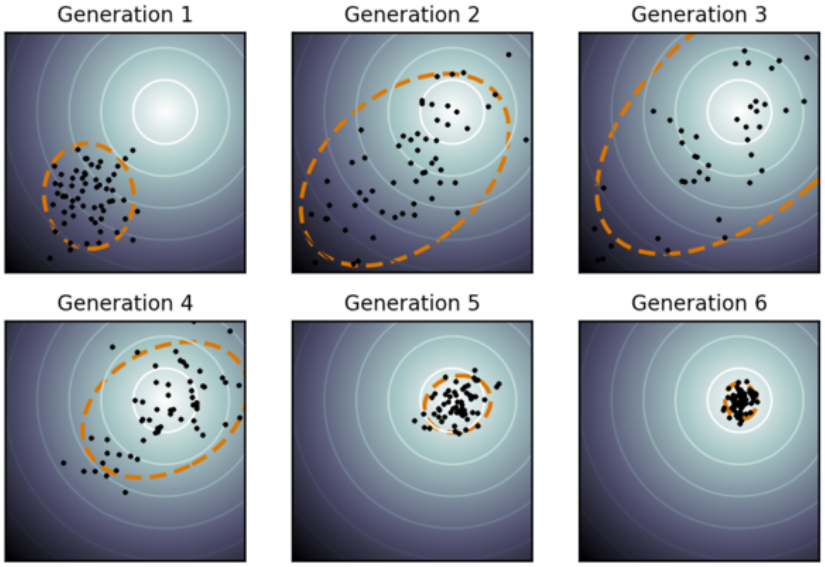

Figure 3: Illustration of the CMA-ES algorithm. The population's loss decreases with each generation, eventually converging to the minimum. Figure taken from [57].

The population at any given step is composed by λ points generated from the multivariate normal distribution

$$x_i^{(g+1)} \sim \mathcal{N}\left(m^{(g)}, \left(\sigma^{(g)}\right)^2 C^{(g)}\right), \tag{40}$$

where the superscripts indicate the generation, m and C are the mean and covariant matrix at a certain iteration and σ is an effective "step size" used to reduce the spread of the population step after step. To generate the next samples from a population, we need to select the best-performing points to evaluate the new mean and covariance matrix. This can be done in several ways. In the papers [53, 54] the strategy was to rank the points from lowest value of the loss function to highest and select the first μ of them. The new mean is then defined through the the weights

$$w_i \propto \mu - i + 1, \qquad \sum_{i=1}^{\mu} w_i = 1, \tag{41}$$

as:

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i x_i^{(g+1)}, \tag{42}$$

where the indexing of the points is such that if $i < j$ then $\mathcal{L}\left(x_i^{(g+1)}\right) < \mathcal{L}\left(x_j^{(g+1)}\right)$. The covariant matrix can be define again employing the weights eq. (41) as

$$C^{(g+1)} = \sum_{i=1}^{\mu} w_i \left(x_i^{(g+1)} - m^{(g)}\right)\left(x_i^{(g+1)} - m^{(g)}\right)^T. \tag{43}$$

This covariant matrix however is a good estimator only if the population is big enough. Empirically this condition is fulfilled when

$$10n \lesssim \left(\sum_{i=1}^{\mu} w_i^2\right)^{-1}, \tag{44}$$

where n is the dimension of x, [53, 54].
In order to have a robust minimization the authors of [53, 54] utilized a similar implementation of Adam for the update of the covariant matrix, where the role of the learning rate was played by the step size σ.

Those papers reported that the CMA-ES strategies resulted to be the most promising optimization algorithm among the ones tested.

3.3 New attempts

In this thesis the "shift" of the fermion previously described was employed. Specifically, let $x \in \mathbb{R}^d$ denote the vector that parameterizes the degrees of freedom of the tensors A_1 and A_2 described in eq. (24). This allows the conditions in eq. (31) to be written as quadratic equations in x. Since there are no terms linear in x, the conditions $\partial V = 0$ can be reformulated as

$$x \cdot T_i x = 0, \quad \forall i = 1, \ldots, N \ , \tag{45}$$

where T_i are known symmetric $\mathbb{R}^d \times \mathbb{R}^d$ matrices. The loss function can be defined as the squared norm of the gradient,

$$\mathcal{L}(x) = \sum_{i=1}^{N} (x \cdot T_i x)^2 \, . \tag{46}$$

During the update procedure however we want to normalize the output to avoid the convergence at the trivial solution $x = 0$. This is the standard SGD approach already discussed in the previous section, which has proven to be effective by papers. However, a significant challenge arises from the presence of local minima in eq. (46). The method proposed here aims to achieve a good initialization $\bar{x}$, increasing the likelihood of convergence to an absolute minimum. To achieve this initialization, two algorithms are introduced. The first algorithm, referred to as the cluster algorithm, is designed to search for "clusters" of points, where each point is a solution to a single equation. The second algorithm, named NS (Near Solution), will be described in a subsequent subsection. The NS algorithm plays a central role in this clustering procedure.

3.3.1 Cluster algorithm

The aim of this algorithm is to identify regions within the parameter space where multiple solutions of the single equations are likely to be found. If we have a small region where there is at least a solution for every equation there is a higher probability to be near a solution respect to a random point, thereby providing a favorable initialization for the SGD algorithm.
If a point x_{sol} is a global minimum of eq. (46), it implies the existence of points $\bar{x}_i$, near x_{sol}, such that $\bar{x}_i \cdot T_i \bar{x}_i = 0$. Each of these points solves at least one of the constraints. The objective is to identify a set of points, all close to each other, where each individual point satisfies the constraint $\bar{x}_i \cdot T_i \bar{x}_i = 0$. The good initialization $\bar{x}$ for the SGD algorithm

is thus defined as the (normalized) average of the points $\bar{x}_i$.
To find these points, the NS algorithm was developed. The objective of the NS algorithm is, given a point x, to find the nearest solution $\bar{x}_i$ such that $\bar{x}_i \cdot T_i \bar{x}_i = 0$. The application of NS is crucial in order to cluster different points near a possible solution. Before explaining the NS algorithm, it is necessary to discuss the procedure that allows for a more accurate estimation of the points $\bar{x}_i$.
Starting from a random guess x, we can use NS to generate the set of points $\{\bar{x}_i\}$. This however does not guarantee proximity to a solution. The idea is to make these points "cluster" as close as possible to a common central point. The procedure begins with a random value $x^{(0)}$ and generate the normalized vectors $\{\bar{x}_i\}$. The mean $x^{(1)}$ of the set $\{\bar{x}_i\}$ is then evaluated, and the most distant element $\bar{x}_j$ from $x^{(1)}$ is identified. Once the element is found we can apply again the algorithm aforementioned to find the vector $\bar{x}'_j$ that is a solution of the corresponding equation and is as close as possible to $x^{(1)}$; this vector is then substituted with $\bar{x}_j$. The mean $x^{(2)}$ of the new set is calculated, and the procedure is iterated as shown in fig. 4. If the mean distance between the vectors decreases significantly over a few iterations, the proposed good initialization is the mean of the set $\{\bar{x}_i\}$. If this is not the case, the procedure is repeated with another random initial value. The testing was done with five clustering steps, where a point was accepted only if the average scalar product between all the $\bar{x}_i$ at the last step were greater than 0.9. These conditions were chosen empirically with little testing, so there is a significant possibility of more effective clustering procedures.

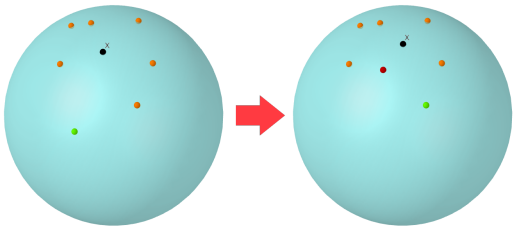

Figure 4: The cyan sphere represents the space of all possible solutions in the simplified case of $d = 3$. The black point represents the normalized average of all the other points, with the green point being the furthest from the average. The red point in the second image is the new value for the green point from the first image.

Data: x
Result: $\bar{x}$
begin
  for $i = 1, \ldots, N$ **do**
    $\bar{x}_i = \mathrm{NS}(T_i,\ x)$
  end
  for $i_{cluster} = 1, \ldots, 5$ **do**
    $\bar{x} = \sum \bar{x}_i / \left|\left| \sum x_i \right|\right|$ **find** j **such that** $\bar{x}_j \cdot \bar{x}$ is minimum $\bar{x}_j = \mathrm{NS}(T_j,\ \bar{x})$
  end
  if $\bar{x}_j \cdot \bar{x} > 0.9$ **then** $\bar{x}$ is returned;
  else repeat the algorithm with a new x;
end

Algorithm 1: Cluster algorithm implementation

3.3.2 NS algorithm

The NS algorithm generates the point $\bar{x}_i$, solution of its respective equation and as close as possible to the initial input x.
The discussion will start with the simplified case where $T_i = D$ is a diagonal matrix, and then extend to the general case. The vectors composed of the positive, zero, and negative eigenvalues of D are denoted as σ_+, σ_0, and σ_-, respectively. This allows the equation $\bar{x}' \cdot D\bar{x}' = 0$ to be written as

$$x_+ \cdot \Sigma_+ x_+ + x_0 \cdot \Sigma_0 x_0 + x_- \cdot \Sigma_- x_- = x_+ \cdot \Sigma_+ x_+ + x_- \cdot \Sigma_- x_- = 0, \tag{47}$$

where $\Sigma = \text{diag}(\sigma)$ and x_+, x_0, and x_- are truncations of $\bar{x}'$. Furthermore, d_+, d_0, and d_- denote the dimensions of the vectors x_+, x_0, and x_-.
Let's consider the following redefinition of the vectors x_+ and x_-,

$$\begin{cases} x_{+,a} \to \sqrt{\sigma_{+,a}} x_{+,a} \\ x_{-,a} \to \sqrt{-\sigma_{-,a}} x_{-,a} \end{cases}, \tag{48}$$

where the index a refers to the component of the respective vector. The eq. (47) can now be rewritten as

$$||x_+||^2 - ||x_-||^2 = 0. \tag{49}$$

The vectors x_+ and x_- can be chosen to have norm 1, while x_0 can have an arbitrarily large or small norm. This parameterizes all the (normalized) solutions of the equation in the space $S^{d_+-1} \oplus \mathbb{R}^{d_0} \oplus S^{d_--1}$ for $d_+, d_- \geq 1$ (when this condition is not met, the number of variables can be reduced, yielding a new system where the condition is met. See the appendix for the discussion of this case).
Concretely we can parameterize the solutions with the vector $u' \in \mathbb{R}^d$ in the following way. We divide the vector u' in three vector u'_+, u'_0 and u'_- with dimensions d_+, d_0 and d_-. We define the new vectors in the following way

$$\begin{cases} u_{+,a} = \frac{1}{\sqrt{\sigma_{+,a}}} \frac{u'_{+,a}}{||u'_+||} \\ u_{-,a} = \frac{1}{\sqrt{-\sigma_{-,a}}} \frac{u'_{-,a}}{||u'_-||} \\ u_0 = u'_0 \end{cases}, \tag{50}$$

and recombine the vectors u_+, u_0 and u_- as u, in a way analogous to the previous decomposition. The solutions of $\bar{x}' \cdot D\bar{x}' = 0$ can then be written as $\bar{x}' = u/||u||$.
To meet the algorithm's requirement that the input vector x' be as close as possible to the output vector $\bar{x}'$, a neural network will be employed, as discussed shortly. Before this, the general case of a non-diagonal T_i matrix will be briefly examined.
Given that the T_i matrices are symmetric, they can always be diagonalized using orthogonal matrices O. This allows the following transformation:

$$\bar{x} \cdot T_i \bar{x} = \bar{x} \cdot ODO^T \bar{x} = (O^T \bar{x}) \cdot D(O^T \bar{x}) = \bar{x}' \cdot D\bar{x}' \tag{51}$$

Thus, passing from the original basis to the diagonal basis requires applying the matrix O^T to the vector, while passing back requires applying the matrix O.

The steps described are summarized in the following diagram.

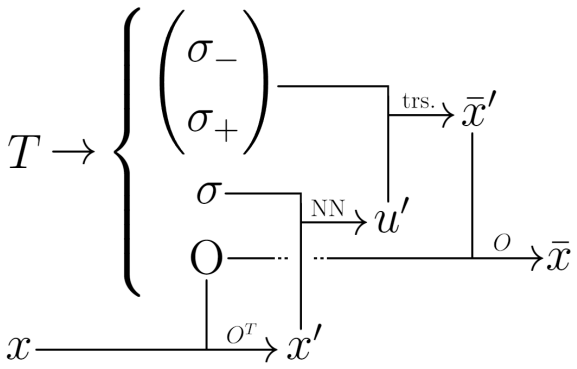

Figure 5: Those are the schematic steps of the NS algorithm to get the vector $\bar{x}$ such that $\bar{x} \cdot T\bar{x} = 0$ and the distance between x and $\bar{x}$ is as low as possible.

Data: O, σ, $(\sigma_+,\ \sigma_-)$, x
Result: $\bar{x}$
begin
$x' = O^T x$
$u' = \mathrm{FFNN}(\sigma,\ x)$
$u' \to (u'_+,\ u'_0,\ u'_-)$
$u_{+,a} = \frac{1}{\sqrt{\sigma_{+,a}}} \frac{u'_{+,a}}{||u'_+||}$
$u_{-,a} = \frac{1}{\sqrt{-\sigma_{-,a}}} \frac{u'_{-,a}}{||u'_-||}$
$u_0 = u'_0$
$(u_+,\ u_0,\ u_-) \to u$
$\bar{x}' = u/||u||$
$\bar{x} = O\bar{x}'$
end

Algorithm 2: NS algorithm implementation

To approximate the unknown function, a standard standard FFNN is sufficient for this purpose.
Since the goal is to minimize the distance between x and $\bar{x}$, a natural choice for the loss function is

$$\mathcal{L} = -x \cdot \bar{x} = -\big(O^T x'\big) \cdot \big(O^T \bar{x}'[u']\big) = -x' \cdot \bar{x}'[u'] \tag{52}$$

where $\bar{x}'[u']$ denotes the vector $\bar{x}'$ parametrized by u'.
The network has as input two d-dimensional vectors, σ and x', and has as output a d-dimensional vector, u'.
Before discussing the layout tested, a comment on the activation functions used is necessary. For the output layer we used the identity function to ensure proper parametrization of the output space. Each layout was tested with different activation functions, with the best performing networks generally using ReLUs exclusively. An exception was a specific layout that used the sigmoid function in one layer, which will be discussed shortly.

The first kind of layouts tested had $2d$ input neurons, fully connected to a certain number of hidden layers, finishing with d output neurons. These layouts were tested with a number of hidden layers ranging from 0 to 4. Among these, the best performance was

observed with 3 hidden layers, containing $2d$, $1.5d$, and d neurons per layer, respectively. Another family of layouts that was tested had a more articulated structure. The first few layers were not fully connected, whereas σ and x' were processed separately. Each input was condensed into an intermediate output using a single hidden layer, with the last neurons employing a sigmoid activation function. These intermediate outputs, along with the vectors σ and x', were then processed through fully connected layers. This architecture proved to be slightly more effective than the previous one, with the idea being that the intermediate outputs could provide additional information useful for better parametrizing the subsequent fully connected layers. The best layout found used an initial hidden layer with $d/2$ neurons per input, condensed into 2 sigmoid outputs. These 4 outputs, together with σ and x', were processed by a single fully connected hidden layer with d neurons before reaching the output layer. The figure below illustrates the two best-performing layouts.

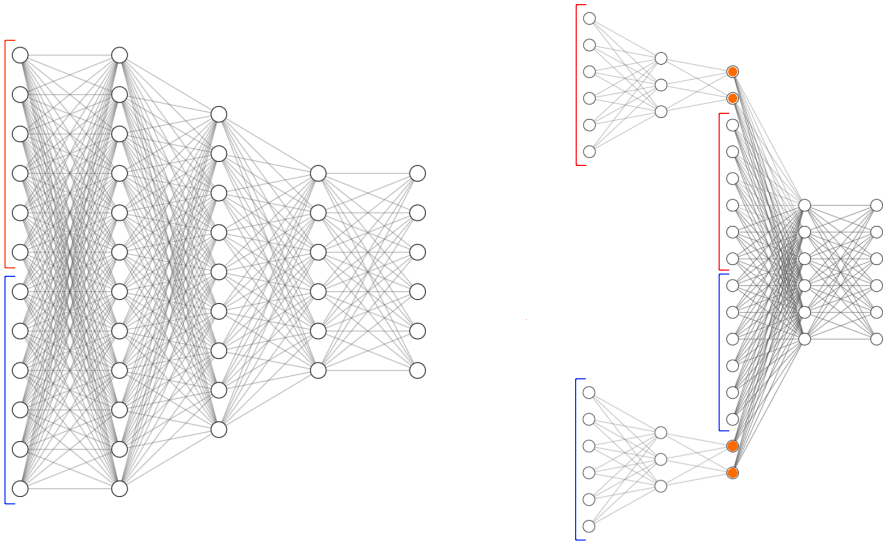

Figure 6: Examples of the best performing layout with $d = 6$. The input neuron are represented with a square bracket, σ in red and x' in blue. All the white neurons, except the output ones, use a ReLU as activation function. The orange neurons use a sigmoid as activation function.

The network training incorporated some simple yet effective techniques to achieve faster and more efficient training. First, while it is possible to develop a network that works with any matrix, during training, only the T_i matrices were used. Although this practice generally risks overfitting, in this context, it is not problematic because these are the only matrices that will serve as input. However, it is worth noting that if another network with the same number of inputs needs to be trained, the initial network would not be suitable for the new task. It is possible, but it was not tested in this thesis, that using the original network as an initialization for the new one could lead to rapid convergence. Another obvious optimization was to evaluate a priori the eigenvalues and orthogonal matrices, since they are the same. This is crucial since their evaluation is considerably slower compared to a forward and backward pass through the network.
For the update procedure, the Adam optimizer was employed, along with an exponential decay of the learning rate.

The described procedure was tested on a system composed by approximately 70 equations with 13 unknowns. Although some solutions were found, the results appeared similar to those obtained using the standard SGD method. The main issue was that, even if it was faster to find the vacua once the network was trained compared to SGD, the overall time was slower. Furthermore for a system with so few variables machine learning implementation were a order of magnitude slower than standard algorithms which produced analytical solutions.
For this reason, the procedure described above was discarded, and non-machine learning approaches were tested. This does not imply, however, that the procedure is invalid in all contexts. As the number of variables increases, traditional algorithms quickly become unfeasible, while machine learning approaches seem to scale better with the number of variables. Additionally, in systems with many solutions, it might be worthwhile to set up the algorithm for initialization instead of proceeding directly with SGD.

The number of parameters in the best-performing network is approximately $4d^2$, so the time complexity of a forward and backward pass is $\mathcal{O}(d^2)$. This implies that the application of this algorithm will scale similarly to the SGD algorithm. However, it is unclear how the network's training time will scale with the number of dimensions. It is likely to scale as $\mathcal{O}(e^d)$, since the entire parameter space for x' needs to be explored. It is worth considering that the transformation to be learned could be relatively simple, such as scaling the x' components based on the eigenvalues σ. Furthermore, in the context of maximal supergravity, most of the eigenvalues are 0. These factors could potentially facilitate the scalability of the training time.

4 Relinearization and new algorithms

In this chapter we will discuss how a system of polynomial equations with many variables and finite solutions can be solved analytically, with the goal of finding every solution. In order to do so, we will first introduce the relinearization algorithm and then a more sophisticated algorithm that builds on it, the XL algorithm. Finally we will present some improvements to the XL algorithm and briefly discuss their implementation in the PyXLTensor library.

4.1 Algebraic approaches

4.1.1 Relinearization algorithm

Finding solutions to multivariate quadratic equations is a problem that appears frequently in many different fields. In cryptography, for instance, a technique known as "relinearization" has been employed to attack data encrypted with the Hidden Field Equations (HFE) scheme [58]. This method was used to solve equations over finite fields but the same algorithm can be used in the case of real parameters. The procedure can be described as follows:
The objective is to solve a set of equations of the form

$$\sum_{i,j=1}^{d} a_{k,ij} x_i x_j = b_k, \qquad k = 1, \dots, N \ , \tag{53}$$

where $a_{k,ij}$ and b_k are known coefficients and $x \in \mathbb{R}^d$ are the unknowns variables. We can introduce the variables $y_{ij} = x_i x_j$, transforming the original system into a set of linear equations. This system however is usually underdetermined and we need to add a constraints on the values of y_{ij}. This additional equations generally are of the form $y_{ij}y_{kl} = y_{ik}y_{jl} = y_{il}y_{jk}$, or higher degree. The new system can then be solved trough standard linearization techniques or applying again the relinearization step. We can show this procedure with the following toy example:

$$\begin{cases} x_1^2 + x_2 x_3 = 7 \\ x_1 x_2 + x_3^2 = 7 \\ x_1 x_2 + x_2 x_3 = 1 \end{cases} . \tag{54}$$

Introducing $y_1 = x_1^2$, $y_2 = x_1 x_2$, $y_3 = x_2 x_3$ and $y_4 = x_3^2$ the above system becomes

$$\begin{cases} y_1 + y_3 = 7 \\ y_2 + y_4 = 7 \\ y_2 + y_3 = 1 \\ y_1 y_3^2 = y_2^2 y_4 \end{cases} , \tag{55}$$

where the last condition is derived by expressing $x_1^2 x_2^2 x_3^2$ in two different ways using the y variables. Using the first three equations, the last can be expressed using only y_3, yielding

$(7-y_3)y_3^2 = (1-y_3)^2(6+y_3)$. The solutions to this equation are $y_3 = -2$, $y_3 = 1/2$ or $y_3 = 3$. Substituting the solutions found for y_3 back in the system we obtain that the possible values for (y_1, y_2, y_3) are $(9, 3, -2)$, $(13/2, 1/2, 1/2)$ or $(4, -2, 3)$. Finally we can solve for the original variables x and get that (x_1, x_2, x_3) is $\pm(3, 1, 2)$, $\pm\sqrt{13/2}(1, 1/13, 1)$ or $\pm(2, -1, -3)$.

4.1.2 XL algorithm

When the system has a small number of variable ($N \lesssim 15$) the procedure described above is an efficient method for finding the solutions. Unfortunately supergravity vacua are usually described by some thousand equations in 912 variables and hence one needs more efficient algorithms to handle bigger systems, like the "XL Algorithm" (eXtended Linearizations) [59]. We will now closely follow the approach presented in the original paper, providing the necessary definitions, explaining the algorithm in detail, and applying it to solve the same example as before using the XL method.

Let K be a field and let $\mathcal{A}$ be a system of multivariate quadratic equations in the form $l_k = 0$ with $k = 1, \ldots, N$.
The equations of the form $l_i \prod_{j=1}^{k} x_{i_j} = 0$ are said to be of the type $x^k l$. For example, the initial equations are of the type l.
x^k denotes the set of all terms of degree k, $\prod_{j=1}^{k} x_{i_j}$.
Let's call $\mathcal{I}_D$ the set of all the polynomials of the type $x^k l$ with total degree less or equal to D $(0 \leq k \leq D-2)$.
The four steps of the algorithm are:

1. **Multiply:** Generate all the terms in $\mathcal{I}_D$.
2. **Linearize:** Consider each monomial of the type x^k as a new variable and perform Gaussian elimination on the equations obtained in 1. The ordering of the monomials must be such that all the terms containing one variable are eliminated last.
3. **Solve:** Assume that step 2 yields at least one univariate equation in the powers of a single variable of x. Solve this equations over K.
4. **Repeat:** Simplify the equations and repeat the process to find the values of the other variables.

Notice the following things. When generating the equations there is no need to consider more general terms such as $l_i^2 = 0$ since they can be obtained as linear combination of elements in $\mathcal{I}_4$. Furthermore it is possible in certain occasions to work only with a subset of all the possible monomials. For example, when all the equations have even (odd) degree terms we can ignore the odd (even) degree monomials.

Let's again consider the system

$$\begin{cases} x_1^2 + x_2x_3 - 7 = 0 \\ x_1x_2 + x_3^2 - 7 = 0 \\ x_1x_2 + x_2x_3 - 1 = 0 \end{cases} \tag{56}$$

and see how the XL algorithm works. For the sake of clarity we will report only the relevant equations.
When performing gaussian elimination on the initial set of equation $\mathcal{I}_2$ we will obtain the first equation in 2 variables, $x_1^2 + x_3^2 - 13 = 0$. In the next iteration, $D = 3$, we obtain another equation involving only x_1 and x_3, namely $x_1^3 - x_3^3 - 7x_1 + 7x_3 = 0$. This equation is obtained from $x_1l_1 = 0$ and $x_3l_2 = 0$. The first of this two equations will generate the even powers of x_3 using x_1, for example

$$\begin{cases} x_1^2x_3^2 + x_3^4 - 13x_3^2 = 0 \\ x_1^4 + x_1^2x_3^2 - 13x_1^2 = 0 \\ x_1^2 + x_3^2 - 13 = 0 \end{cases}, \tag{57}$$

that during gaussian elimination gives us $x_3^4 - x_1^4 + 26x_1^2 - 169 = 0$. During the next steps we will leave the even powers of x_3 explicit even if during the gaussian elimination they could be simplified.
At the step $D = 4$ some of the equations are

$$\begin{cases} x_1^3x_3 - x_3^4 - 7x_1x_3 + 7x_3^2 = 0 \\ x_1^4 - x_1x_3^3 - 7x_1^2 + 7x_1x_3 = 0 \\ x_1^3x_3 + x_1x_3^3 - 13x_1x_3 = 0 \end{cases}. \tag{58}$$

At $D = 6$ those equations generates

$$\begin{cases} x_1^5x_3 - x_1^2x_3^4 - 7x_1^3x_3 + 7x_1^2x_3^2 = 0 \\ x_1^4x_3^2 - x_1x_3^5 - 7x_1^2x_3^2 + 7x_1x_3^3 = 0 \\ x_1^6 - x_1^3x_3^3 - 7x_1^4 + 7x_1^3x_3 = 0 \\ x_1^4x_3 - x_1^2x_3^4 - 7x_1^3x_3 + 7x_1^2x_3^2 = 0 \\ x_1^5x_3 + x_1^3x_3^3 - 13x_1^3x_3 = 0 \end{cases}. \tag{59}$$

After performing gaussian elimination at the step $D = 6$ we obtain one equation that involves only x_1,

$$-12x_1^6 + 234x_1^4 - 1446x_1^2 + 2808 = 0. \tag{60}$$

By solving this equation and substituting back in the system the result, we obtain all the solutions, like in the previous example.
Although this algorithm may appear more computationally expensive, it is expected to perform with polynomial complexity in n when dealing with overdefined systems. In particular, as discussed in [59], the algorithm is likely to succeed when $D \approx \lceil 1/\sqrt{\varepsilon} \rceil$, with $N = \varepsilon d^2$ and a total complexity of $\mathcal{O}(n^{\omega/\sqrt{\varepsilon}})$, where ω is a constant approximately in the range of $2 \div 3$, depending on the efficiency of the algorithm used for gaussian elimination.

4.2 Improvements on the XL algorithm

The XL algorithm is a very efficient algorithm for solving systems of quadratic equation, however, its performance decreases as the number of variables and equations increases. In the context of supergravity, empirically, the algorithm generally becomes impractical on standard machines when the number of variables reaches the magnitude of 30. In this section, we will discuss various improvements that could increase the efficiency of the algorithm. The algorithm is implemented in Python and is available as open-source under the PyXLTensor library [60] (a detailed technical discussion is provided in the appendix). This library allows users to define known and unknown tensors, specify high and low indices, and establish the relative metrics. Tensors can be summed, contracted, or generated based on their block structure, all in a physicist-friendly manner. Once the equations are set up, the library can be used to solve for the unknown parameters using the enhanced version of the XL algorithm, which we will now discuss.

The first difference from the previously discussed implementation of XL is that we are interested in finding all possible solutions to a system. Each time an equation is solved, a new system is created for each equation, and the solutions are substituted into these systems.
The other, more substantial, difference lies in the reduction process. In the standard implementation of XL, a gaussian elimination is carried out. However, this reduction does not account for the goal of obtaining equations with the fewest number of variables. In order to introduce a ranking system for comparing equations, we first need to define the concept of "priority" among monomials. The terminology used from this point forward is propaedeutic for understanding the code, with square brackets [...] referring to the common name of various objects or the names of implemented functions.

We will say that a monomial [`var`] has priority [`System.__has_priority(var1, var2)`] over another monomial if:

- The degree of the first monomial is greater than the degree of the second monomial.
- If the degrees are the same, the powers of the variables of the monomials are compared in order (from x_1 to x_d). The monomial with the larger exponent is said to have priority.

The leading variable [`leading_var`] of an equation [`eq`] is the monomial with the biggest priority.
An equation will rank higher than another one if:

- The first equation contains fewer distinct variables than the second one.
- If they contain the same number of distinct variables, the equation with fewer terms will rank higher.
- If they have the same number of terms, the equation with the highest priority will rank higher.

The priority condition is essential to avoid stalling situations. This can be demonstrated with a simple example:

$$\begin{cases} x^2 + y^2 = 1 \\ x^2 + z^2 = 2 \\ y^2 + z^2 = 3 \end{cases} \tag{61}$$

Without the priority condition, this system would not simplify, as any combination of two equations will always involve at least two equations and three monomials.

To simplify two equations, we check all possible linear combinations that cancel at least one monomial. If the linear combination with the highest priority outranks one of the two starting equations, it substitutes the equation with the lower priority.
The reduction process [`System.reduce(self)`] can be made efficient by sorting the equations by priority [`System.__sort(self)`]. We begin with an empty list of equations. Starting with the equation of the highest priority in the original set, we attempt to reduce it using every other equation in the new list. Every time a reduction occurs, the equation with the highest priority is inserted into the new list, while the equation with the second-highest priority is used for further comparisons. By filling the new list in this way, we maintain a sorted set and avoiding redundant comparisons.

We now describe how a step [`Solver.__step(self)`] of the algorithm is executed in this implementation.
From the set containing all systems that need to be processed [`Solver.list_systems`], the system with the fewest equations is selected.
If the degree of the system exceeds a predetermined value, the current solution is saved among the undetermined solutions [`Solver.Undetermined_Solutions`]. If this is not the case, the system is analyzed to find new (partial) solutions [`System.find_new_Sol(self)`]. If the system does not yield any new solutions [`to_reduce`], it will be reduced. If new solutions are found, the new system will undergo the procedure just described until no further solutions are identified. At this point, all systems will be reduced as previously discussed.
If a system undergoes reduction without any changes to its equations, it implies that a step up in degree [`System.step_up(self)`] is necessary, as in the standard XL implementation. However, if the system's equations change, the step-up procedure may not be required, and it is therefore avoided. All the new systems, if present, are added to the initial list of systems to be processed and the step procedure can be done again, as shown in the following diagram.

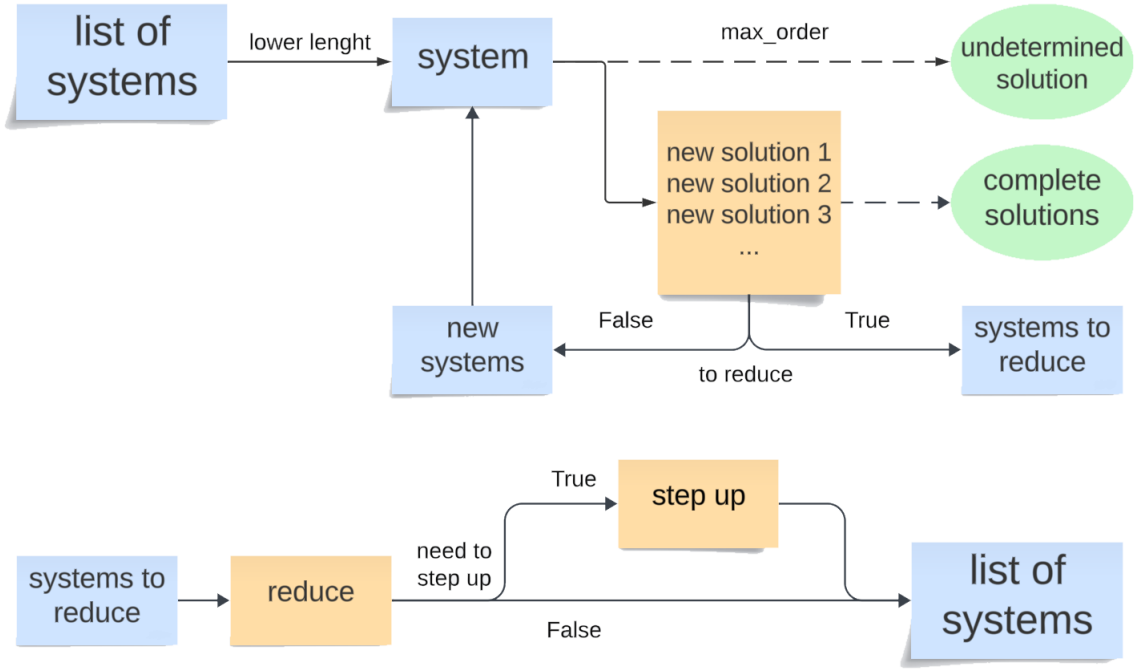

Figure 7: Flow chart of the enhanced XL algorithm.

The full algorithm consists of repeating the step procedure until no systems remain, either because the solutions have been found or because the maximum allowed degree [`max_order`] is reached. This final condition can occur in two scenarios: either the degree required to solve the system exceeds the maximum degree that has been set, or the system has parametric solutions. In either case, the partial solutions are saved and can be substituted back into the original system for further analysis by other means.

4.3 Examples

As already shown in eq. (24), the T-tensor can be expressed using the tensors A_1 and A_2. Furthermore these tensors can be chosen to be invariant under the action of the elements of a group G_g, ensuring that the resulting theory has G_g as the gauge group. The consistency equations and can be written as follows:

$$\begin{aligned}
0 =& A^k{}_{lij}A_n{}^{mij} - A_l{}^{kil}A^m{}_{nij} - 4A^{(k}{}_{lni}A^{m)i} - 4A_{(n}{}^{mki}A_{l)i} \\
&- 2\delta^m_l A_{ni}A^{ki} + 2\delta^k_n A_{li}A^{mi}, \\
0 =& A^i{}_{jk[m}A^k{}_{npq]} + A_{jk}\delta^i_{[m}A^k{}_{npq]} - A_{j[m}A^i{}_{npq]} \\
&+ \frac{1}{24}\varepsilon_{mnpqrstu}\left(A_j{}^{ikr}A_k{}^{stu} + A^{ik}\delta^r_j A_k{}^{stu} - A^{ir}A_j{}^{stu}\right), \\
0 =& A^r{}_{ijk}A_r{}^{mnp} - 9A^{[m}{}_{r[ij}A_{k]}{}^{np]r} - 9\delta^{[m}_{[i}A^n{}_{|rs|j}A_{k]}{}^{p]rs} \\
&- 9\delta^{[mn}_{[ij}A^{|u|}{}_{k]rs}A_u{}^{p]rs} + \delta^{mnp}_{ijk}A^u{}_{rst}A_u{}^{rst}.
\end{aligned} \tag{62}$$

Additionally, the extremization of the scalar potential, given in eq. (31), is expressed as:

$$\begin{aligned} 0 =& \mathcal{C}_{ijkl} + \frac{1}{24}\varepsilon_{ijklmnpq}\mathcal{C}^{mnpq}, \\ \mathcal{C}_{ijkl} =& A^{m}{}_{[ijk}A_{l]m} + \frac{3}{4}A^{m}{}_{n[ij}A^{n}{}_{kl]m}, \end{aligned} \tag{63}$$

as shown in [27]. Here the normalizations are such that $\varepsilon_{12345678} = -1$ and $\delta^{1,\ldots,n}_{1,\ldots,n} = \frac{1}{n!}$.

Before solving for new parametrizations of the tensors A_1 and A_2, the PyXLTensor library was tested on families of A_1 and A_2 for which all solutions were known. In particular the G_2 truncation, SU(4) truncation and SU(3) truncation reported on [23] were tested and all the solutions were successfully found.

The systems of equations generated for the SU(4) and SU(3) cases were processed using the PyXLTensor library. Unfortunately, the limitations of the current hardware prevented these systems from being solved. A supercomputer could be employed to solve these equations, as similar-sized systems have been solved using the standard implementation of the XL algorithm.
Solving these systems is particularly interesting, as there currently exists no complete classification of the vacua associated with these gaugings.

5 Summary and outlooks

In this thesis we discussed maximally symmetric vacua for maximal supergravity in 4D. We summarized the procedure that allows us to study the properties of a vacua through the embedding tensor and its parametrizations. Two promising algorithms were presented: one based on machine learning techniques and the other based on more traditional computational methods. Additionally we introduced modifications to these algorithms that allows them to be more effective. Lastly we used the new implementation of the XL algorithm in an attempt to completely classify the vacua generated by the gaugings of SU(4) and SU(3).
Unfortunately, due to limitations in computational resources we were not able to perform those calculations. The classification of vacua for even smaller gauge groups, such as SU(2), or the most general cases with no a priori structure, is not feasible given our current understanding of maximal supergravity and the available computing power.
Beyond solving the systems of equations for the gaugings of SU(4) and SU(3) on more powerful hardware, there are other improvements that can be done, both algorithmically and mathematically. The PyXLTensor library can be improved in various way, including a potential shift to more efficient programming languages like C++. One interesting problem, highlighted in Appendix B, is that currently we do not know any efficient method to linearly combine indefinite matrices to produce, if possible, a positive definite matrix. If this problem could be solved, it would allow for the reduction of the number of variables in systems of quadratic equations, such as those arising in the context of maximal supergravity.
Although we were unable to compute these new vacua, the algorithmic contributions of this thesis can still be relevant in their respective fields. In particular, the PyXLTensor library can be an effective tool for solving systems of polynomial equations, derived by tensor expressions, that are parametrized by many variables.

A The group $E_{7(7)}$

The group $E_{7(7)}$ is a 133−dimensional group with 63 compact generators and 70 non-compact generators. The main representations that will be used in this thesis are the fundamental **56** and the adjoint **133** representations, in the following we will discuss two different choices for the generators in the **56** representation.
The most simple way to define the elements of $\mathfrak{e}_{7(7)}$ in the fundamental representation is by imposing constraints on the elements of $\mathfrak{sp}(56,\mathbb{R})$,

$$\begin{pmatrix} \Lambda_{AB}{}^{CD} & \Sigma_{ABEF} \\ \star\Sigma^{GHCD} & \Lambda'^{GH}{}_{EF} \end{pmatrix} \tag{64}$$

with

$$\begin{aligned} \lambda_A{}^B &= -\lambda'^B{}_A, \quad \lambda_A{}^A = 0, \\ \Lambda_{AB}{}^{CD} &= 2\lambda_{[A}{}^{[C}\delta_{B]}{}^{D]}, \\ \star\Sigma^{ABCD} &= \frac{1}{4!}\varepsilon^{ABCDEFGH}\Sigma_{EFGH}. \end{aligned} \tag{65}$$

Here the indices are pair wise antisymmetrized and Λ and Σ are 28×28 matrices. The matrices $\lambda_A{}^B$ and $\lambda'^B{}_A$ belong respectively to the **8** and **8'** representations of SL(8) and therefore $\Lambda_{AB}{}^{CD}$ and $\Lambda'^{AB}{}_{CD}$ belong respectively to the **28** and **28'** representations of SL(8). This base makes manifest the decomposition under SL(8)

$$\mathbf{56} \to \mathbf{28} \oplus \mathbf{28'}. \tag{66}$$

The maximal compact subgroup of $E_{7(7)}$ is H =SU(8)/$\mathbb{Z}_2$. In maximal supergravity, the coset group $E_{7(7)}/H$ is of particular importance, for this reason we are interested in finding a basis where the SU(8) generators are evident.
Imposing on top of the previous relations

$$\begin{aligned} \lambda_A{}^B &= -\lambda_B{}^A, \\ \Sigma_{ABCD} &= -\star\Sigma^{ABCD}, \end{aligned} \tag{67}$$

The total number of parameters is equal to $\binom{8}{2} + \frac{1}{2}\binom{8}{4} = 63$ and since imposing this constraint give us that the generators of eq. (64) are compact, those are the generators of SU(8).
In order to better see the SU(8) generators we can introduce complex coordinates, i.e. we can represent the vector (x_{AB}, y^{CD}) as $x_{AB} \pm iy^{AB}$, the infinitesimal transformation can then be written as

$$\begin{aligned} \delta(x_{AB} \pm iy^{AB}) = (\Lambda_{AB}{}^{CD} \pm i\star\Sigma^{ABCD})(x_{CD} \pm iy^{CD}) = \\ = (\Lambda_{AB}{}^{CD}x_{CD} + \Sigma_{ABCD}y^{CD}) \pm i(\star\Sigma^{ABCD}x_{CD} - \Lambda_{CD}{}^{AB}y^{CD}). \end{aligned} \tag{68}$$

The elements in the coset $E_{7(7)}/H$ are not generated by matrices that belong to the base obtained with eq. (67), so we must impose the following conditions on eq. (64)

$$\begin{aligned} \lambda_A{}^B &= \lambda_B{}^A, \\ \Sigma_{ABCD} &= \star\Sigma^{ABCD}. \end{aligned} \tag{69}$$

As before, we can represent the vector (x_{AB}, y^{CD}) using complex coordinates and the infinitesimal transformation for this generators can be written as

$$\begin{aligned}\delta(x_{AB} \pm iy^{AB}) = (\Lambda_{AB}{}^{CD} \pm i \star \Sigma^{ABCD})(x_{CD} \mp iy^{CD}) = \\ = (\Lambda_{AB}{}^{CD} x_{CD} + \Sigma_{ABCD} y^{CD}) \pm i(\star\Sigma^{ABCD} x_{CD} - \Lambda_{CD}{}^{AB} y^{CD}).\end{aligned} \tag{70}$$

The action of compact and noncompact generators can be written using a single formula introducing the vector $(z, \bar{z})$, with $z = x + iy$ and defining the matrices

$$\begin{aligned}\lambda &= \hat{\Lambda}_{AB}{}^{CD} + i \star \hat{\Sigma}^{ABCD}, \\ \sigma &= \tilde{\Lambda}_{AB}{}^{CD} + i \star \tilde{\Sigma}^{ABCD},\end{aligned} \tag{71}$$

where $\hat{.}$ refers to the matrices used for the compact generators in eq. (67) while $\tilde{.}$ refers to the matrices used for the noncompact generators in eq. (69); the infinitesimal transformation of an element under $E_{7(7)}$ is

$$\begin{pmatrix}\delta z \\ \delta \bar{z}\end{pmatrix} = \begin{pmatrix}\lambda & \sigma \\ \bar{\sigma} & \bar{\lambda}\end{pmatrix} \begin{pmatrix}z \\ \bar{z}\end{pmatrix}. \tag{72}$$

In order to exchange between the two bases that we discussed we can introduce the matrix

$$S_{\underline{M}}{}^{N} = \frac{\sqrt{2}}{8} \begin{pmatrix}\Gamma_{AB}{}^{CD} & i\Gamma_{ABCD} \\ \Gamma^{ABCD} & -i\Gamma^{AB}{}_{CD}\end{pmatrix}. \tag{73}$$

Here the matrices Γ are defined starting from the Clifford algebra Cliff(8)

$$\{\Gamma^A, \Gamma^B\} = -2\delta^{AB}\mathbb{1}, \tag{74}$$

and antisymmetrizing them

$$\Gamma^{ABCD} = \Gamma^{[A}\Gamma^B\Gamma^C\Gamma^{D]}. \tag{75}$$

The underline index is used for SU(8) covariant index and the index without the underline is used for SL(8) covariant index instead. More concretely the transformation between the SL(8) and SU(8) bases is

$$[t_\alpha]_{\underline{M}}{}^{\underline{N}} = S_{\underline{M}}{}^{P} [t_\alpha]_P{}^{Q} S^\dagger_Q{}^{\underline{N}}. \tag{76}$$

B How to deal with matrices with 0 positive(negative) eigenvalue

In section 3.3 we presented a particular initialization for SGD algorithms to solve a system of equations of the form

$$x \cdot T_i x = 0, \quad \forall i = 1, \ldots, N\ . \tag{77}$$

In the section we present an algorithm that works for matrices with at least 1 positive and 1 negative eigenvalue, we will now discuss how to proceed if this is not the case.

Let suppose that the matrix T has not negative eigenvalues (the other case is analogous). As before we can rotate x and go in a base where the matrix T assumes the form $D = \text{diag}(0, \dots, 0, \sigma_+)$, where σ_+ is the vector of positive eigenvalues. If we truncate x in x_0 and x_+ as we done before, the equation to solve becomes

$$x_+ \cdot \text{diag}(\sigma_+)x_+ = 0, \tag{78}$$

that has $x_+ = 0$ has its only solutions.
This means that each time a matrix T has only positive or negative eigenvalues we can rotate all the matricides in the diagonal base for this particular matrix. Eliminate all the rows and columns corresponding to the non zero eigenvalues of D for each matrix. This will give us a system with a number of parameter equal to the initial one minus the number of positive eigenvalues of T. This new system could contain redundant equation or even more matricides with only positive or negative eigenvalues, in the latter case the procedure described can be repeated.
Once a solution for this system is found it sufficient to add 0 in the spots that were previously removed an rotate back the vector in order to be a solution of the original system.

One could try to find a linear combination of matrices that results in a matrix with only positive or negative eigenvalues. Although this is possible, We have not been able to find, either in the literature or through calculations, methods to find such combinations. This is something that could be further studied because it is a great opportunity to quickly reduce the complexity of the system.

C How to use PyXLTensor

The library PyXLTensor can be found here [60].

C.1 Initialization of the tensors

Tensors can be initialized in two different ways, by directly specifying all elements or by defining the tensor shape. The other parameters necessary to the initialization are the tensor name, its indices and the metrics related to the various indices, if they are different from the identity.
tensor_name (string): Name of the tensor
indices_string (string): String containing all the indices in order, if an index name is preceded by ^ is an upper index, if it is preceded by _ is a lower index.
tensor=None (list, tuple or numpy.array): If given, specifies the values of the tensor.
shape=None (list, tuple or numpy.array): If no tensor is given, specifies the dimensions of the tensor.
metrics=None (list, tuple or numpy.array): Lists the metric tensors associated with each index. If unspecified, the identity metric is assumed for each index.

```
import PyXLTensor as xlt

"""
the tensor t1 has two indices, alpha (up) and beta (down), the value of
    the tensor is known and all the metrics are the identity.
the tensor t2 has two indices, beta (up) and gamma (down), the value of
    the tensor is unknown and the metric associated with gamma is not
   the identity.
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma', shape=(2, 2), metrics=[[[1, 0],
   [0, 1]], [[-1, 0], [0, 1]]])
```

The tensor name is important to keep track of the unknown tensors. It's fundamental that unknown tensors do not share the same name.

C.2 Basic operations

The PyXLTensor library allows for tensor arithmetic using the standard symbols. For example additions and subtractions can be performed directly with + and -. These operations require that the indices of the tensors are consistent with each other. They can be in different orders as long as there are indices that share the same name, size, metric and are both up or both down.

```
import PyXLTensor as xlt

"""
Correct:
Even if the indices are not in the same order they have the same name,
   size, positioning an metrics.
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '_beta^alpha', shape=(2, 2))
t3 = -t2 + t1
```

```
import PyXLTensor as xlt

"""
Wrong, respectively:
(t1) Mismatch of the name of the indices.
(t2) Mismatch of the position of the indices.
(t3) Mismatch of the dimention of the indices.
(t4) Mismatch of the metrics of the indices.
"""
t = xlt.Tensor('t', '^alpha_beta', tensor=[[1, 2], [0, -1]])

t1 = xlt.Tensor('t2', '^gamma_beta', shape=(2, 2))
t_sum = t + t1

t2 = xlt.Tensor('t2', '^alpha^beta', shape=(2, 2))
t_sum = t + t2
```

```
t3 = xlt.Tensor('t2', '^alpha_beta', shape=(2, 3))
t_sum = t + t3

t4 = xlt.Tensor('t2', '^alpha_beta', shape=(2, 2), metrics=[[[1, 0],
    [0, 1]], [[-1, 0], [0, 1]]])
t_sum = t + t4
```

PyXLTensor also supports multiplication and division by a scalar using * and /, respectively, and tensor contractions via @. When contracting tensors one has to make sure that the contracted indices are one up and the other down and share the same properties.

```
import PyXLTensor as xlt

t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma', shape=(2, 2), metrics=[[[1, 0],
    [0, 1]], [[-1, 0], [0, 1]]])
t3 = 2 * t1 @ t2
```

The library includes methods for summing and contracting lists of tensors, respectively `Tensor.sum_all` and `Tensor.contract_all`. All tensor rules of these operations must be respected like if those operations were performed one by one. If a string is provided after the list of tensors, the string will be the tensor name of the new tensor.

```
import PyXLTensor as xlt

"""
Sum of tensors
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^alpha_beta', tensor=[[3, -1], [1, 1]])
t3 = xlt.Tensor('t3', '^alpha_beta', tensor=[[2, 1], [-4, 3]])
t4 = xlt.Tensor.sum_all([t1, t2, t3], 't4')

"""
Contraction of tensors
"""
t1 = xlt.Tensor('t1', '^alpha_beta', tensor=[[1, 2], [0, -1]])
t2 = xlt.Tensor('t2', '^beta_gamma^i', tensor=[[[3, -1], [1, 1]], [[0,
    -2], [3, 2]]])
t3 = xlt.Tensor('t3', '^gamma_alpha', tensor=[[2, 1], [-4, 3]])
t4 = xlt.Tensor.contract_all([t1, t2, t3], 't4')
```

C.3 Block tensors

PyXLTensor supports the combination of tensor lists into a single block tensor by specifying the indices to be grouped. In order to specify which indices will be combined and how, we will give a list, `grouping_of_indices`. The elements of this list will be another list, with two elements, the first one is the name for the new index in the block tensor, the other is the list of the indices name that will be grouped together. All the other indices

will be untouched and must coincide. Consider the following example:

$$T^{I}{}_{J}{}^{K}{}_{L} = \begin{pmatrix} a^{i_1}{}_{j_1}{}^{K}{}_{L} & b^{i_1}{}_{j_2}{}^{K}{}_{L} & c^{i_1}{}_{j_3}{}^{K}{}_{L} \\ d^{i_2}{}_{j_1}{}^{K}{}_{L} & e^{i_2}{}_{j_2}{}^{K}{}_{L} & f^{i_2}{}_{j_3}{}^{K}{}_{L} \end{pmatrix} \tag{79}$$

with $I = (i_1, i_2)$ and $J = (j_1, j_2, j_3)$. As always indices with the same names must have the same proprieties (size, up or down, metric) but the order of them does not matter. We can implement this tensor in the following way

```
import PyXLTensor as xlt

"""
Block tensor
This is an 8x8x8x8 tensor, even if the indices are not in the same
    order the sizes are consistent among the tensors
"""
a = xlt.Tensor('a', '^i1^K_j1_L', shape=(2, 8, 1, 8))
b = xlt.Tensor('b', '^i1_L^K_j2', shape=(2, 8, 8, 3))
c = xlt.Tensor('c', '^K_L_j3^i1', shape=(8, 8, 4, 2))
d = xlt.Tensor('d', '^i2_j1^K_L', shape=(6, 1, 8, 8))
e = xlt.Tensor('e', '^K^i2_L_j2', shape=(8, 6, 8, 3))
f = xlt.Tensor('f', '^K^i2_L_j3', shape=(8, 6, 8, 4))
grouping_of_indices = [['I', ['i1', 'i2']],
                       ['J', ['j1', 'j2', 'j3']]]
T = xlt.Tensor.block_tensor([a, b, c, d, e, f], grouping_of_indices)
```

The order of the tensor in the list is not important and he name of the grouped index can be arbitrary and it doesn't have to be of the type $[name1,\ name2, \ldots,\ name\#n]$.

C.4 Symmetrization, anti-symmetrization, duality, δ and ϵ

Given a tensor it is possible to perform the (anti-)symmetrization of some indices. Those indices must be all upper indices or all lower indices, have the same size and the same metric. The normalization is such that the (anti-)symmetrization of (anti-)symmetric indices will result in the same tensor.
If a tensor is entered manually the code will not check if there are (anti-)symmetric indices. In general imposing the symmetrization on those tensors can help the code run faster when dealing with big tensors.

```
import PyXLTensor as xlt

"""
(Anti-)Symmetrization
"""
t = xlt.Tensor('t', '^alpha^beta', tensor=[[1, 2], [0, -1]])
t_sym = t.symmetrize('alpha', 'beta')
t_asym = t.anti_symmetrize('alpha', 'beta')
```

Another operation that can be done is the duality operation. Given a set of indices the duality operation is equivalent to the contraction with a Levi-Civita tensor with the given indices and the division by the factorial of the number of contracted indices. The indices of the epsilon tensor are all lower (upper) if the contracted indices are all up (down). The convention used for the sign of the Levi-Civita tensor is $\epsilon^{1,\dots,n} = \epsilon_{1,\dots,n} = +1$.

```
import PyXLTensor as xlt

"""
Duality
"""
t = xlt.Tensor('t', '^c', tensor=[1, -1, 0])
star_t = t.dual('a', 'b', 'c')
```

The Kronecker delta δ and Levi-Civita ϵ tensors are in-build functions and can be defined as expected, the delta takes as input the indices, given with the same form for the initialization of a normal tensor and the dimension of the two indices, while the epsilon only needs the indices.

```
import PyXLTensor as xlt

"""
Delta and epsilon tensors
"""
d8 = xlt.Tensor.delta('^i_j', 8)
epsilon = xlt.Tensor.epsilon('_a_b^c')
```

C.5 Managing the indices

Indices can be raised (lowered) using the method `.to_raise` (`.to_lower`) giving the indices to be raised (lowered). The operation of raising and lowering the indices is done through the metric. The metric is the tensor used to lower the indices, the inverse metric is calculated to raise the indices.
This method will return another tensor with the new indices, while the original tensor is left untouched.

```
import PyXLTensor as xlt

"""
Raising and lowering indices
"""
t = xlt.Tensor('t', '^i_j_k^l', shape=(2, 2, 2, 2), metrics=[[[-1, 0],
    [0, 1]], ] * 4)
t_all_up = t.to_raise('j', 'k')
t_all_down = t.to_lower('i', 'l')
```

There are two different ways to change the name of some indices.
The first one makes use of the method `Tensor.change_indices_name`, this method will have two lists as inputs, the first list will have the old names of the indices and the second one will have the corresponding new names.
The second one needs you to specify all the (old and) new names of the indices in order. It can be done by simply putting square brackets after the tensor and specifying all the new names in the correct order.
The internal order of the indices of the tensor can be seen by calling the attribute `.indices`. This ordering can be changed by simply using the method `Tensor.set_order_indices` and giving the new desired order. Notice that while all the other methods always give you a new tensor, independent from the original one, this method simply modifies the attributes of the object without returning anything.

```
import PyXLTensor as xlt

"""
Changing the indexing
"""
t = xlt.Tensor('t', '^a^j^c^l', shape=(2, 2, 2, 2))
print(t.indices)  # ['a', 'j', 'c', 'l']
t1 = t.change_indices_name(['j', 'l'], ['b', 'd'])
print(t1.indices)  # ['a', 'b', 'c', 'd']
t2 = t['a', 'b', 'c', 'd']
print(t2.indices)  # ['a', 'b', 'c', 'd']
t2.set_order_indices('d', 'c', 'b', 'a')
print(t2.indices)  # ['d', 'c', 'b', 'a']
```

$t1$ and $t2$ are the same tensor, even if we change the order of the indices of $t2$, so while after those operation $t1 + t2$ is still a valid operation $t + t1$ is not.

When relabeling the indices, if an upper index and a lower index happens to have the same name, a trace will be automatically performed.

C.6 Elements of the tensors

In order to keep track of the unknown variables in the tensors the class `Poly_Expression` is used, the elements of the tensor are instances of this class.
`Poly_Expression` objects support standard operations with the symbols +, -, * and / (notice however that the division can be used only to divide by a scalar). The expression of a `Poly_Expression` object is determined by its only attribute, `.sum_variables`.
`Poly_Expression.sum_variables` is a list that contains all the summed monomials of an expression, a empty list is associated with the value 0. The monomials are a list of two elements, a overall constant that multiplies the unknowns and the list of the product of the single variables. A variable is itself a list of two objects, the name of the tensor from which it is taken and a tuple with the indices values associated with that variable.
The `Poly_Expression` objects are handled entirely by the `Tensor` class so it is not necessary to create or modify instances of this class directly.

We can clarify better the structure of `Poly_Expression.sum_variables` with the following example:
If we define the two by two tensor x and we took the trace, the resulting element (of the 0-dimensional `Tensor`) is $x_{00} + x_{11}$, written as:

$$\Big[\big[1, [\text{'x'}, (0, 0)]\big], \big[1, [\text{'x'}, (1, 1)]\big]\Big].$$

The square of the previous expression is $x_{00}^2 + 2x_{00}x_{11} + x_{11}^2$, written as:

$$\Big[\big[1, [\text{'x'}, (0, 0)], [\text{'x'}, (0, 0)]\big], \big[2, [\text{'x'}, (0, 0)], [\text{'x'}, (1, 1)]\big], \big[1, [\text{'x'}, (1, 1)], [\text{'x'}, (1, 1)]\big]\Big].$$

C.7 Reading the tensors

One way to get the elements of the tensor is to use the square brackets like it is done to change the indices name. When using the square bracket every numerical element will be taken as the value of the respective index (the index can range from 0 to the size of the index minus one). If all the entries given are numeric, the respective element of the tensor will be returned, if the elements are both numeric and strings the corresponding subtensor will be returned.
Tensor can be shown by simply printing the object, the printing follows the same rules of a `numpy.array`.
The tensor can be obtained by calling the attribute `.tensor`, however this is a `numpy.array` of `Poly_Expression` and thus can be impractical to use. In the case of fully determined tensors the `numpy.array` composed by all the numerical entries of the tensor can be obtained by the method `.get_numeric_tensor`.

```
import PyXLTensor as xlt

"""
Getting the elements of the tensor
"""
t = xlt.Tensor('t', '^a^b', tensor=[[1, 2], [0, -1]])
print(t[0, 1])  # 2 (Poly_Expression)
print(t['a', 0])  # t^{a0} = (1, 0)^{a0} (Tensor)
print(t[1, 'c'])  # t^{1c} = (0, -1)^{1c} (Tensor)
print(t.get_numeric_tensor())  # [[1, 2], [0, -1]] (numpy.array)
```

C.8 Initializing a system of equations

PyXLTensor is intended for writing equations of the type $T = 0$, where T is a tensor. To generate a list of equations, first of all it is necessary a list containing all the unknown tensors. If the complete system of equation is generated by different tensor expression, the list of the unknown tensors must be the same for every set of equations (the order of the tensors mus be the same too). The method that returns this set of equations is `Tensor.get_equations(unknown_tensors)`.

The `Solver` class is used to read and process the systems equations. It can be initialized by simply giving the list of equations and the maximum degree of the polynomial that will be processed.

```
import PyXLTensor as xlt

"""
Write the equations and initialize the System.
"""
v = xlt.Tensor('v', '_a', shape=(3, ))
t = xlt.Tensor('t', '_a', shape=(3, ))
M = xlt.Tensor('M', '^a^b', tensor=[[0, 1, 0], [1, 0, 0], [0, 0, -1]])
u = xlt.Tensor('u', '_b', tensor=[0, 1, 1])
one = xlt.Tensor('', '', tensor=1)

Cond1 = M @ v['b'] + t.to_raise('a')
Cond2 = v @ M @ u
Cond3 = v @ t.to_raise('a') - one
Cond4 = v.to_raise('a') @ v - one

unknown_tensors = [v, t]
list_equations = []
list_equations += Cond1.get_equations(unknown_tensors)
list_equations += Cond2.get_equations(unknown_tensors)
list_equations += Cond3.get_equations(unknown_tensors)
list_equations += Cond4.get_equations(unknown_tensors)

system = xlt.System(list_equations, 4)
```

C.9 Obtaining and reading the solutions

In order to avoid floating-point errors when dealing with expressions that should be zero, there is the possibility to set the tolerance for the value 0. Each value whose magnitude is less than this threshold will be considered zero.
To obtain the solutions is sufficient to run `System.get_Solutions()`. This method can take as input the maximum number of step tried, this can be used as a safe measure if the systems take to long to be processed. By default, if no argument is given there will me no maximum number of steps.
This method will return two list, the first one will contain all the complete solutions while the second will contain all the partial solutions. The solutions can be turned back in tensor form by using the method `Tensor.get_tensors_from_Sol(unknown_tensors, Sol)`. This method takes as input the unknown tensors, with the same order used when generating the equations, and a solution from one of the two list. The method will return the unknown tensors where all the known entries are substituted with their respective values.

```
# continuation of the previous code

xlt.zero_tolerance = 1e-8  # default value 1e-12

Complete_Solutions, Undetermined_Solutions = system.get_Solutions()
C_Tensor_Solutions = [xlt.Tensor.get_tensors_from_Sol(unknown_tensors,
    Sol) for Sol in Complete_Solutions]
U_Tensor_Solutions = [xlt.Tensor.get_tensors_from_Sol(unknown_tensors,
    Sol) for Sol in Undetermined_Solutions]

print('Solutions to tensor equations')
for i, Tensor_Sol in enumerate(C_Tensor_Solutions + U_Tensor_Solutions)
    :
     print(f'Solution {i + 1}:\n')
     for tensor in Tensor_Sol:
         print(tensor)
```

References

[1] B. de Wit, H. Samtleben and M. Trigiante, *The Maximal D=4 supergravities*, *JHEP* **06** (2007) 049 [0705.2101].

[2] B. de Wit, H. Samtleben and M. Trigiante, *On Lagrangians and gaugings of maximal supergravities*, *Nucl. Phys. B* **655** (2003) 93 [hep-th/0212239].

[3] G. Dall'Agata and G. Inverso, *On the Vacua of N = 8 Gauged Supergravity in 4 Dimensions*, *Nucl. Phys. B* **859** (2012) 70 [1112.3345].

[4] F. Englert, *Spontaneous compactification of eleven-dimensional supergravity*, *Physics Letters B* **119** (1982) 339.

[5] B. De Wit and H. Nicolai, *N = 8 supergravity*, *Nuclear Physics B* **208** (1982) 323.

[6] B. de Wit and H. Nicolai, *The Parallelizing S(7) Torsion in Gauged N = 8 Supergravity*, *Nucl. Phys. B* **231** (1984) 506.

[7] N.P. Warner, *Some New Extrema of the Scalar Potential of Gauged N = 8 Supergravity*, *Phys. Lett. B* **128** (1983) 169.

[8] N.P. Warner, *Some Properties of the Scalar Potential in Gauged Supergravity Theories*, *Nucl. Phys. B* **231** (1984) 250.

[9] C.M. Hull, *The Minimal Couplings and Scalar Potentials of the Gauged N = 8 Supergravities*, *Class. Quant. Grav.* **2** (1985) 343.

[10] C.M. Hull and N.P. Warner, *The Structure of the Gauged N = 8 Supergravity Theories*, *Nucl. Phys. B* **253** (1985) 650.

[11] C.M. Hull and N.P. Warner, *The Potentials of the Gauged N = 8 Supergravity Theories*, *Nucl. Phys. B* **253** (1985) 675.

[12] C.M. Hull and N.P. Warner, *Noncompact Gaugings From Higher Dimensions*, *Class. Quant. Grav.* **5** (1988) 1517.

[13] C.M. Hull, *New gauged N=8, D = 4 supergravities*, *Class. Quant. Grav.* **20** (2003) 5407 [hep-th/0204156].

[14] K. Behrndt and M. Cvetic, *General N = 1 supersymmetric flux vacua of (massive) type IIA string theory*, *Phys. Rev. Lett.* **95** (2005) 021601 [hep-th/0403049].

[15] D. Lust, F. Marchesano, L. Martucci and D. Tsimpis, *Generalized non-supersymmetric flux vacua*, *JHEP* **11** (2008) 021 [0807.4540].

[16] T. Fischbacher, *Fourteen new stationary points in the scalar potential of SO(8)-gauged N=8, D=4 supergravity*, *JHEP* **09** (2010) 068 [0912.1636].

[17] T. Fischbacher, *Numerical tools to validate stationary points of SO(8)-gauged N=8 D=4 supergravity*, *Comput. Phys. Commun.* **183** (2012) 780 [1007.0600].

[18] T. Fischbacher, K. Pilch and N.P. Warner, *New Supersymmetric and Stable, Non-Supersymmetric Phases in Supergravity and Holographic Field Theory*, 1010.4910.

[19] T. Fischbacher, *The Encyclopedic Reference of Critical Points for SO(8)-Gauged N=8 Supergravity. Part 1: Cosmological Constants in the Range -\Lambda/g^2 \in [6:14.7)*, 1109.1424.

[20] G. Dall'Agata, G. Inverso and M. Trigiante, *Evidence for a family of SO(8) gauged supergravity theories*, *Phys. Rev. Lett.* **109** (2012) 201301 [1209.0760].

[21] H. Kodama and M. Nozawa, *Classification and stability of vacua in maximal gauged supergravity*, *JHEP* **01** (2013) 045 [1210.4238].

[22] A. Borghese, A. Guarino and D. Roest, *All G_2 invariant critical points of maximal supergravity*, *JHEP* **12** (2012) 108 [1209.3003].

[23] A. Borghese, G. Dibitetto, A. Guarino, D. Roest and O. Varela, *The SU(3)-invariant sector of new maximal supergravity*, *JHEP* **03** (2013) 082 [1211.5335].

[24] F. Catino, G. Dall'Agata, G. Inverso and F. Zwirner, *On the moduli space of spontaneously broken $N = 8$ supergravity*, *JHEP* **09** (2013) 040 [1307.4389].

[25] A. Borghese, A. Guarino and D. Roest, *Triality, Periodicity and Stability of SO(8) Gauged Supergravity*, *JHEP* **05** (2013) 107 [1302.6057].

[26] G. Dall'Agata, G. Inverso and A. Marrani, *Symplectic Deformations of Gauged Maximal Supergravity*, *JHEP* **07** (2014) 133 [1405.2437].

[27] A. Gallerati, H. Samtleben and M. Trigiante, *The $\mathcal{N} > 2$ supersymmetric AdS vacua in maximal supergravity*, *JHEP* **12** (2014) 174 [1410.0711].

[28] Y. Pang, C.N. Pope and J. Rong, *Holographic RG flow in a new $SO(3) \times SO(3)$ sector of ω-deformed $SO(8)$ gauged $\mathcal{N} = 8$ supergravity*, *JHEP* **08** (2015) 122 [1506.04270].

[29] A. Guarino, D.L. Jafferis and O. Varela, *String Theory Origin of Dyonic N=8 Supergravity and Its Chern-Simons Duals*, *Phys. Rev. Lett.* **115** (2015) 091601 [1504.08009].

[30] A. Guarino and O. Varela, *Dyonic ISO(7) supergravity and the duality hierarchy*, *JHEP* **02** (2016) 079 [1508.04432].

[31] I.M. Comsa, M. Firsching and T. Fischbacher, *SO(8) Supergravity and the Magic of Machine Learning*, *JHEP* **08** (2019) 057 [1906.00207].

[32] A. Guarino, J. Tarrío and O. Varela, *Halving ISO(7) supergravity*, *JHEP* **11** (2019) 143 [1907.11681].

[33] N. Bobev, T. Fischbacher, F.F. Gautason and K. Pilch, *New AdS_4 Vacua in Dyonic ISO(7) Gauged Supergravity*, 2011.08542.

[34] P. Karndumri and C. Maneerat, *Janus solutions from dyonic ISO(7) maximal gauged supergravity*, *JHEP* **10** (2021) 117 [2108.13398].

[35] D. Berman, T. Fischbacher, G. Inverso, B. Scellier and B. Scellier, *Vacua of ω-deformed SO(8) supergravity*, *JHEP* **06** (2022) 133 [2201.04173].

[36] L.-G. Gagnon, *Machine learning for track reconstruction at the lhc*, *Journal of Instrumentation* **17** (2022) C02026.

[37] L. Wang, *Discovering phase transitions with unsupervised learning*, *Phys. Rev. B* **94** (2016) 195105.

[38] G. Kanwar, M.S. Albergo, D. Boyda, K. Cranmer, D.C. Hackett, S. Racanière et al., *Equivariant flow-based sampling for lattice gauge theory*, *Phys. Rev. Lett.* **125** (2020) 121601 [2003.06413].

[39] S. Ruder, *An overview of gradient descent optimization algorithms*, *arXiv preprint arXiv:1609.04747* (2016) .

[40] J. Jordan, *"Setting the learning rate of your neural network"*, https://www.jeremyjordan.me/nn-learning-rate/, 2018.

[41] K. Santosh, N. Das and S. Ghosh, *Chapter 2 - deep learning: a review*, in *Deep Learning Models for Medical Imaging*, K. Santosh, N. Das and S. Ghosh, eds., Primers in Biomedical Imaging Devices and Systems, pp. 29–63, Academic Press (2022), DOI.

[42] I. Sutskever, J. Martens, G.E. Dahl and G.E. Hinton, *On the importance of initialization and momentum in deep learning*, in *International Conference on Machine Learning*, 2013, https://api.semanticscholar.org/CorpusID:10940950.

[43] J. Duchi, E. Hazan and Y. Singer, *Adaptive subgradient methods for online learning and stochastic optimization*, *Journal of Machine Learning Research* **12** (2011) 2121.

[44] G. Hinton, *Neural networks for machine learning*, *https : //www.cs.toronto.edu/ tijmen/csc*321*/slides/lecture_slides_lec*6*.pdf*.

[45] D.P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[46] J. Schmidhuber, *Deep learning in neural networks: An overview*, *Neural Networks* **61** (2015) 85–117.

[47] R. Quiza and J.P. Davim, *Computational methods and optimization*, pp. 177–208 (2011), DOI.

[48] V. Kunc and J. Kléma, *Three decades of activations: A comprehensive survey of 400 activation functions for neural networks*, 2024.

[49] A.G. Baydin, B.A. Pearlmutter, A.A. Radul and J.M. Siskind, *Automatic differentiation in machine learning: a survey*, 2018.

[50] K. Hornik, M. Stinchcombe and H. White, *Multilayer feedforward networks are universal approximators*, *Neural Networks* **2** (1989) 359.

[51] N. Bobev, T. Fischbacher, F.F. Gautason and K. Pilch, *A cornucopia of AdS_5 vacua*, *JHEP* **07** (2020) 240 [2003.03979].

[52] C. Krishnan, V. Mohan and S. Ray, *Machine Learning $\mathcal{N} = 8, D = 5$ Gauged Supergravity*, *Fortsch. Phys.* **68** (2020) 2000027 [2002.12927].

[53] D. Partipilo, *New methods for old problems: vacua of maximal $D = 7$ supergravities*, *JHEP* **09** (2022) 096 [2205.06245].

[54] D.F. Partipilo, *(Machine) Learning Supergravity Vacua*, Ph.D. thesis, U. Padua (main), 2022.

[55] L.M. Schmitt, *Theory of genetic algorithms*, *Theoretical Computer Science* **259** (2001) 1.

[56] N. Hansen, *The cma evolution strategy: A tutorial*, 2023.

[57] J. Chaquet, *Solving Differential Equations with Evolutionary Algorithms*, Ph.D. thesis, 07, 2015.

[58] A. Kipnis and A. Shamir, *Cryptanalysis of the hfe public key cryptosystem by relinearization*, in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, ed., (Berlin, Heidelberg), pp. 19–30, Springer Berlin Heidelberg, 1999.

[59] N. Courtois, A. Klimov, J. Patarin and A. Shamir, *Efficient algorithms for solving overdefined systems of multivariate polynomial equations*, pp. 392–407, 05, 2000, DOI.

[60] R. Costantini, *PyXLTensor*, 2024.