Modelling of magnetic structure using interior point optimization given exchange parameters

Arthur C. Ronner

Bachelor project
TU Delft

Delft, June 15, 2022
Supervisors: Prof. Dr. E. Brück
Dr. C. Urzúa Torres
Dr. N. van Dijk
Dr. I. Batashev

Abstract

In this report, a model is presented to alleviate some of the computational work that goes into the effort of finding the magnetic properties of magnetocaloric materials. The model utilizes an interior point optimization routine to solve for the minimal exchange energy configuration of a system, given the exchange interactions of the material. The model is tested against four materials (Ni, MnO, Fe_2P and Mn_2Sb). For Ni and MnO, the exchange interactions are also computed. Three iterations of the model are compared. The base model, which only considers exchange interactions inside a chosen supercell, the base model with the inclusion of boundary conditions, and the base model with boundary conditions and the addition of an algorithm to find optimal solutions.

The algorithm analyzes the found results by the optimization routine, and if the result is considered not properly symmetric, runs the optimization routine another time, from a symmetrical starting point obtained from the outcome of the previous run.

In all versions of the model, effectiveness (percent of runs that resulted in the optimal configuration) and average run times were recorded. Three initialization methods for the model were used, and also tested for their effectiveness. For the algorithm, a parameter γ is introduced that changes the size of some of the moments for the new starting points. Six different values for γ were tested for their effectiveness against a test set of suboptimal solutions. The model with the addition of boundary conditions and the algorithm performed the best out of the three iterations of the model, with an effectiveness of 99.895%, and an average run time ranging from 0.62 s for $2\times2\times2$ Ni, to 94.64 s for $3\times3\times3$ Fe_2P, in the case of $\gamma = 0.3$. To conclude, the model with the inclusion of the boundary conditions and the algorithm proves to be a robust method to evaluate the magnetic configuration of a material, especially for smaller systems.

Contents

1. Introduction

Magnets truly are curious materials. Natural magnets have already persisted throughout history for a long period of time, but have mostly only been used as a compass for the majority of time since their discovery. It was only after world war 2 that a large surge in research on magnetic materials really started to kick in [1, pp. xiii–xv]. A large reason for this change was the discovery of neutron diffraction, a groundbreaking discovery that allowed for the closer study of individual atoms in a material.
This new rise of interest in magnets, also gave rise a great amount of applications. Not the least of which is their usage in computers, which sparked even more research, applications, and so the cycle continued.
More recently a lot of research is conducted in the field of magnetocaloric materials [2] [3], magnetic materials that, when their magnetic configuration is altered, heat up or cool down. These materials have the potential to play a large role in the energy transition, as they can be utilized as an alternative for current refrigerators, which utilize harmful gasses in their core cooling process [4]. Current research concerning magnetocaloric materials is still very much in the exploration phase, and the search for the best material for this application is still very much a work in process [5].
As there is an extreme amount of different potential materials, this search is an endeavor in which a lot of testing has to be done per material considered. This means that, in the search for these new materials, there are a lot of computationally intensive steps to take before the required information about a given material is known.
Therefor, in this study, an attempt is made to simplify this process, specifically, the finding of the magnetic properties of a material. A model is proposed that, when the exchange interactions of a material are known; models the magnetic structure of the material by utilizing the interior point optimization routine. The model is subsequently tested to a number of materials (Ni, MnO, Fe_2P and Mn_2Sb), to test its effectiveness at predicting a magnetic structure. Three iterations of the model will be discussed; the base model, the model with the inclusion of boundary conditions, and the model with boundary conditions and a search algorithm.
This report will be structured in the following manner. First; the underlying theory supporting the model will be discussed in section 2. This will include numerous concepts from the field of magnetic materials, as well as numerical solving methods, and material properties for the materials listed above. Next, in section 3, the methodology of the model and the data acquisition for the utilized materials will be covered. This will include the methodology of each of the iterations of the model listed above. Subsequently, the results of the model and the material data acquisition will be presented in section 4, and the discussion and the conclusions in sections 5 and 6, respectively. Lastly, appendices A through D will include a 3D image of the boundary conditions, a flow chart for the model, auxiliary results and the python code of the model, respectively.

2. Theory

A number of closely related, separate pieces of theory are used as a basis for the model described in the next chapter. First, magnetism and related topics are discussed, followed by some computational methods, mathematical concepts, and finally some information regarding materials used in this project.

2.1. Types of magnetism

Materials can be magnetic in a couple of different ways. The types of magnetism considered here are the ones relevant to this project. Mainly, these types of magnetism can be subdivided into either a ordered (in about 13 different ways) or disordered state. A material is called disordered (paramagnetic) when its atoms have a random fluctuating magnetic moment in the absence of an external magnetic field, resulting in a net zero magnetic moment. As soon as a magnetic field is applied; they align with the external field[1].
A material can also be ordered (see figure 1). A material is called ferromagentic when all magnetic moments of atoms point in the same direction, and thereby creating a large magnetic field, even in absence of an external magnetic field. Antiferromagnetic materials are ordered, but their moments exactly cancel each other. Lastly (for the scope of this project;) we have ferrimagnetism, where like an antiferromagnet, moments point in opposite directions, but like a ferromagnet, does induce a net moment. This is due to the different sizes of moments. [1, pp. 87–197]

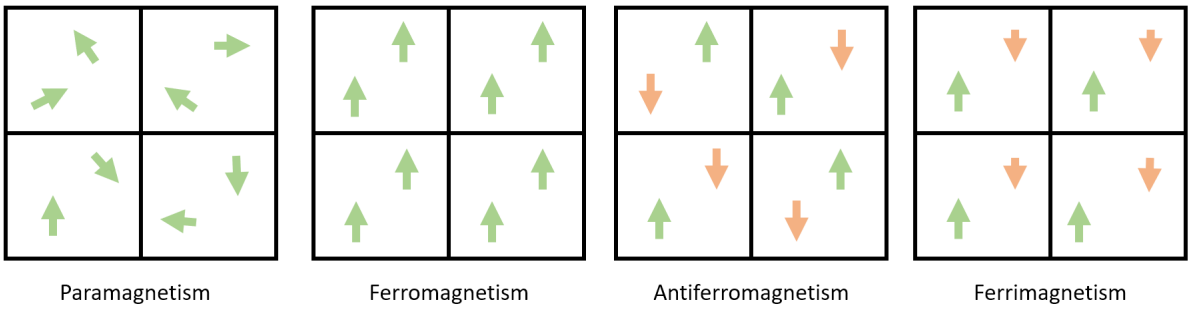


Figure 1: 2D example of the different types of magnetism. Magnetic moments of different atoms are represented by arrows. Red and green is used to emphasize direction when applicable.

Note that the above examples are not a comprehensive list, as more exotic structures also exist (for instance spin spirals, among others).
Each 'ordered' magnetic material, only has this ordered property up until a certain temperature, at which the material becomes paramagnetic. This is because a material is ordered as this is its lowest energy configuration. As a material heats up however, atoms gain more and more thermal motion (entropy), thereby getting less and less incentive to align their magnetic moments as the magnetic contribution to the total energy of the atom is less and less significant [5]. Each magnetically ordered material has a distinct magnetic phase transition (called the Curie (T_c) or Néel (T_N) temperature for ferro- and antiferromagnetic materials respectively) above which the material loses its magnetic ordering.
One may wonder why some materials are differently ordered magnetically than others, or about the origin of the magnetic order of materials. For that, we will have to look at exchange interactions.

[1] In general, though the temperature of a material also has to be considered. If the material is too warm, it may not be able to be pushed into order. If the material is cold enough, it may already be ordered without an external field.

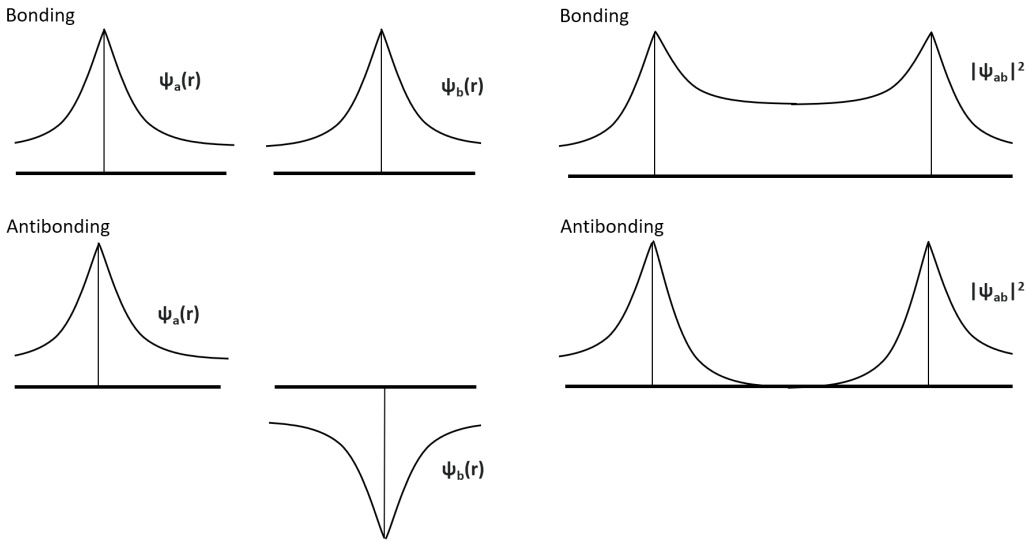


Figure 2: Different positional options for two electrons. ψ_a, ψ_b and ψ_{ab} represent the wave functions or electron a, b and their combined wavefunction respectively.

2.2. Exchange interactions

The exchange interactions are a atomic scale property of materials that couple the direction of the magnetic moments of different atoms in its crystal structure. It exists because of the principle of indistinguishability of quantum particles and the Pauli exclusion principle in the context of the electrons of a material. Indistinguishability tells us that if two identical quantum mechanical particles have an overlapping location probability, and one particle is measured in the overlapping area, there is no way of telling which one of the two was measured. Then note that the Pauli exclusion principle tells us that two fermions have an antisymmetric wavefuntion with respect to exchange [6]. From this, it is clearly shown that the two interacting electrons have two options for their wavefunctions. Either their positional wavefunction is symmetric and their spin wavefuction is antisymmetric, or vice versa (as their total wavefunction can be considered a product of the two). If we then look at the combined system, note that both cases result in a different energy for the total system as the ideal position for the electrons with respect to their atoms is different (see figure 2), and hence the energy associated with its position is altered. This interaction is called the exchange interaction (1).

$$J = E_{\uparrow\uparrow} - E_{\uparrow\downarrow} \tag{1}$$

Where J is the exchange interaction, and $E_{\uparrow\uparrow}$ and $E_{\uparrow\downarrow}$ are the energies for parallel and antiparallel alignment, respectively. Crucially, which of the two states is optimal depends on the system. The energy associated with the exchange interaction can now be described by equation 2:

$$E = -2J_{kl}\mathbf{s}_k \cdot \mathbf{s}_l \tag{2}$$

Where s_k and s_l are the spin vectors of electron k and electron l, respectively, and J_{kl} is the exchange interaction between the two. From this, we can derive a total exchange energy for a system of multiple atoms, if we simplify the individual contributions of electrons into one exchange interaction per atom. Note that the exchange interaction is the only term that we evaluate in the Hamiltonian of the system.

$$E = -2\sum_{i \neq j} J_{ij}\mathbf{S}_i \cdot \mathbf{S}_j \tag{3}$$

Where $\mathbf{S}_i$ and $\mathbf{S}_j$ are the magnetic moments of atoms i and j, and J_{ij} is the exchange interaction between the two. Where the factor 2 is a convention, though sometimes written as 1/2 or 1 in some sources [7]. The minus sign is also up to convention. In this work a minus sign and a prefactor of 2 is used, following the most common notation [8, 9, 10].

2.3. Magnetocaloric effect

The magnetocaloric effect (MCE), in general, is defined as a magnetic material heating or cooling due to the removal or activation of a magnetic field [11]. The MCE is especially strong at a phase transition (most commonly that of the Curie or Néel temperature), as then, the largest difference in magnetic field over the smallest difference in temperature is present. This effect is interesting, as it allows the heating or cooling of a material by changing the external magnetic field in a energy efficient manner, something that can be exploited in a number of applications.
Quantitatively, the MCE can be explained by a thermodynamic process. Consider a paramagnetic material under adiabatic conditions, i.e. a state in which the total entropy of the system is constant, for instance due to isolation of the material. The disordered magnetic state results in a high entropy [12]. If a magnetic field is applied, the magnetic configuration becomes ordered, and therefore loses part of its entropy. As the total entropy of the material has to remain constant, the material is heated up.
More explicitly, it can be derived in formulas in the following way, using the Maxwell relations in equations (4) and (5) [12].

$$\left(\frac{\partial S_M(T,H)}{\partial H}\right)_T = \left(\frac{\partial M(T,H)}{\partial T}\right)_H \tag{4}$$

Where S_M is the entropy due to magnetization M, T is temperature, H is external magnetic field, and M is internal magnetic field.

$$\left(\frac{\partial S(T,H)}{\partial T}\right)_H = \left(\frac{C(T,H)}{T}\right)_H \tag{5}$$

Where S is the total entropy in the system, and C is the heat capacity at constant pressure. For both equations, T and H in the bottom right corner of a derivative indicates that said quantity is kept constant. Equation (4) relates the difference in entropy at a constant temperature to the change in magnetization at a constant external field. Equation (5) signifies the entropy change due to temperature T, at a constant external magnetic field H. Combining these two expressions with the expansion of the derivative of S shown in (6), where we use that the entropy of the system is constant (so $dS = 0$), we obtain the following result (7).

$$TdS = T\left(\frac{\partial S(T,H)}{\partial T}\right)_H dT + T\left(\frac{\partial S(T,H)}{\partial H}\right)_T dH \tag{6}$$

$$dT(T,H) = -\left(\frac{T}{C(T,H)}\right)_H \left(\frac{\partial M(T,H)}{\partial T}\right)_H dH \tag{7}$$

Finally, we integrate the obtained result with respect to H to obtain (8).

$$\Delta T_{ad}(T)_{\Delta H} = \int_{H_I}^{H_F} dT(T,H) = -\int_{H_I}^{H_F} \left(\frac{T}{C(T,H)}\right)_H \left(\frac{\partial M(T,H)}{\partial T}\right)_H dH \tag{8}$$

Where ΔT_{ad} is the adiabatic temperature difference, depending on temperature and the change in H. H_I and H_F are the initial and final external fields, respectively. Equation (8) gives an expression for the magnetocaloric effect. From (8), we can notice that the magnetocaloric effect dependent on temperature and the external magnetic field. Secondly, it is also material dependent, and not easily calculated using first principles [12].

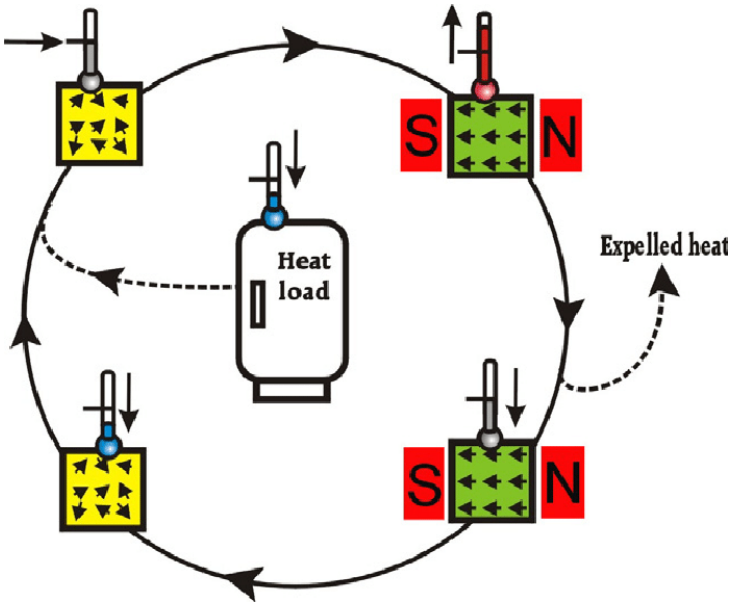


Figure 3: The magnetocaloric refrigeration cycle. Starting from the top left; activation of the magnetic field aligns and heats up the material. Then, heat is expelled, to cool down the material. Next, the external field is turned off, and the temperature of the material drops significantly. Lastly; the material extracts heat from the source, cooling down the source in the process. Image retrieved from [3].

It should also be noted, that given this expression using the change in entropy, it is also more evident that the magnetocaloric effect is largest at a magnetic phase transition, as the the change in entropy is relatively large, resulting in a large temperature change.

2.3.1. Applications

As per time of writing; there are three main applications/areas of research for the magnetic effect. Two of them operate near room temperature (refrigeration, and heat pumps), and one operates near the absolute zero, called adiabatic demagnitization. The latter is the only of the three that is actively used already [4]. The first two will be discussed further.

Refrigeration In principle, the magnetocaloric refrigeration method functions the same as a standard vapour compression devices, where the difference is that magnetocaloric refrigeration utilizes a magnetocaloric material and a (rotating) magnet attached to an electric motor as a driving force [4], where gas refrigeration uses the expanding and compression of gas (see figure 3 for the magnetocaloric refrigeration cycle). The gasses utilized for this purpose are however quite harmful for the environment [2], and their efficiency limit has been reached. Hence, this is an area where the magnetocaloric alternative has a lot of advantages over its predecessor (namely a higher efficiency and less noise while operating), although most applications are still on the expensive or experimental side [4].

Heat pumps Apart from providing a sustainable solution to classical refrigerators, for heat pumps, magnetocaloric materials can add to a quickly developing, already sustainable field [13]. For heat pumping, a thermodynamic cycle is used to extract heat from a reservoir, and transport it to the desired location. Most well known are the geothermal heat pumps, that have their reservoir well below the ground, and transport heat to a location at the surface. Magnetic materials are one of the many possibilities for the further development of this popular technology [14]. Another application is using this heat pumping process reversely; by turning the excess heat of industrial processes into cheap electricity [4].

2.4. Magnetocrystalline anisotropy

Magnetocrystalline anisotropy (MCA) is an effect related to the magnetic moment of an atom where, in contrast to the exchange parameters, the energy contribution generated by the magnetic field to a Hamiltonian is dependent on the direction in a crystal. Usually, these directions correspond to the miller indices of the crystal structure. For example: (100) and (110) could have a different MCA, thereby changing the orientation of the magnetic field of the material along the axis where the MCA enery is minimal. In general, the MCA energy is a very small contribution compared to the exchange interaction. It is therefor also hard to determine, especially using first principles[2] [1]. For illustration; the following approximation can be made to determine the MCA energy:

$$E = K_0 + K_1\left(\alpha_1^2\alpha_2^2 + \alpha_2^2\alpha_3^2 + \alpha_3^2\alpha_1^2\right) + K_2\left(\alpha_1^2\alpha_2^2\alpha_3^2\right) + \cdots \tag{9}$$

Where E is the MCA energy, K_i are the anisotropy constants, and α_i are the angles between the direction of the magnetic field and the crystal axes. The anisotropy constants are the reason that the first principle determination is hard, as they depend on the spin orbit coupling, which is in general not easily described by first principles. A general theory and methodology is known however, called the force theorem. A reasoning and proof behind this method can be found in [15]. In more recent years, numerous papers have been released using a first principles method to describe the MCA energy ([16] and [17] for example). However, those methods are still in early stages and limited to the field of hard magnets, which mainly focus on rare earth metals (which are typically not readily available). As the magnetocrystalline anisotropy usually only constitutes a small energy difference compared to the exchange interactions, and it is difficult to determine using first principles, it will be disregarded in the model.

2.5. Density-functional theory

Density-functional theory (DFT) is a computational method to compute the Schrödinger equation for many body systems. DFT approximates the solution of the Schrödinger equation. This is extremely useful in practice, as it simplifies computation and makes solutions for other materials than the hydrogen atom possible. It usually outputs these solutions in the form of Bloch waves. Bloch waves are one of the ways to express the solution to a Schrödinger equation for a periodic structure. They are used as the solution Schrödinger equation in a crystal structure. They can be seen as a plane wave modulated by a periodic function. In this project, the Viena Ab initio simulation package (VASP) was used as an implementation of DFT, as a part of the calculation of exchange interactions of a material.

2.6. Wannier functions

Wannier functions, just as Bloch waves, are a solution to the Schrödinger equation in crystalline structures. Wannier functions differ from Bloch waves in that they describe the system locally [18]. When a set of Bloch waves is known, a set of Wannier functions can be constructed using the next equations. A standard Bloch wave can be described as follows:

$$\psi_{\mathbf{k}}(\mathbf{r}) = e^{i\mathbf{k}\cdot\mathbf{r}}u_{\mathbf{k}}(\mathbf{r}) \tag{10}$$

Where $\psi_{\mathbf{k}}(\mathbf{r})$ represents Bloch wave with index $\mathbf{k}$, corresponding to a reciprocal lattice point. $\mathbf{r}$ represents the location in real space, and $u_{\mathbf{k}}$ is a function with the periodicity of the lattice in each lattice direction. Then, a Wannier function is defined as the following:

$$\phi_{\mathbf{R}}(\mathbf{r}) = \frac{1}{\sqrt{N}}\sum_{\mathbf{k}} e^{-i\mathbf{k}\cdot\mathbf{R}}\psi_{\mathbf{k}}(\mathbf{r}) \tag{11}$$

[2]First principles, or *ab initio*, is the term in physics to describe that something is calculated purely from theory, without relying on empirical data.

Where $\phi_{\mathbf{R}}(r)$ is the Wannier function corresponding to real location $\mathbf{R}$, N is the total number of Bloch waves, and $\mathbf{R}$ represents the centers of the orbitals being described. The locality of the Wannier functions is achieved by the complex exponential function, as is common in the translation from reciprocal to real space.

2.7. Green's function method

The Green's function method is a general way to solve differential equations, using Green's functions. It is presented here as it is used in the general procedure of finding the exchange interactions of a material. Given a general differential equation with a source term, the Green function can be seen as the response of the system to an impulse function, as illustrated by the following example.

$$\frac{d^2x}{dt^2} + \omega^2 x = f(t); \quad x(0) = 0, x'(0) = 0 \tag{12}$$

Where x is the unknown function, w is the frequency of the system and $f(t)$ is a source term. Notice, that $f(t)$ can also be written as an integral over δ-functions without loss of generality.

$$f(t) = \int_0^\infty f\left(t'\right) \delta\left(t' - t\right) dt' \tag{13}$$

Then, the Greens function is defined as follows:

$$\frac{d^2G\left(t,t'\right)}{dt^2} + \omega^2 G\left(t,t'\right) = \delta\left(t' - t\right); \quad G\left(0,t'\right) = 0, G'\left(0,t'\right) = 0. \tag{14}$$

Where $G(t,t')$ is the Greens function, dependent on time (t) and a moment t'. Notice that G is defined to satisfy the same boundary conditions as x. Given this definition for G, one can now use G to solve for x in the following manner.

$$x(t) = \int_0^\infty G\left(t,t'\right) f\left(t'\right) dt' \tag{15}$$

Where one can easily check that the previous equation holds, applying 12 to 15. This gives:

$$\frac{d^2x(t)}{dt^2} + \omega^2 x(t) = \int_0^\infty \left[\frac{d^2G\left(t,t'\right)}{dt^2} + \omega^2 G\left(t,t'\right)\right] f\left(t'\right) dt' \tag{16}$$

Which is easily worked out to be the required solution (using equation 14). Now, the problem has been simplified into finding G, with which x can be determined. Solving G can be a lot of work on its own, and the mathematics behind this procedure differ from problem to problem. In some cases it is easier than finding x directly, which is why the method is utilized. [19, pp. 394–459]
A more specific version of the Greens function method is used in the field of solid state physics. This method is called the Korringa, Kohn and Rostoker (KKR) greens function method, to its inventors, who specifically used the Green's function method in solving the Schrödinger equation of a lattice system. The methodology is the same as above, as the Schrödinger equation can be rewritten in the form described in equation (17).

$$\begin{aligned} &\left[-\nabla^2 + V(\mathbf{r}) - E\right] \psi(\mathbf{r}) = 0 \\ &\psi\left(\mathbf{r} + \mathbf{r}_\mathrm{o}\right) = \exp\left(i\mathbf{k} \cdot \mathbf{r}_s\right) \psi(\mathbf{r}) \end{aligned} \tag{17}$$

Where $V(\mathbf{r})$ is a periodic potential, $\mathbf{r}$ is the real space location, and k is the reciprocal space location. $\mathbf{r_s}$ is a translation vector in real space. Then, $\psi(\mathbf{r})$ can be described in the following form [20]:

$$\psi(\mathbf{r}) = \int_\tau G\left(\mathbf{r},\mathbf{r}'\right) V\left(\mathbf{r}'\right) \psi\left(\mathbf{r}'\right) d\tau' \tag{18}$$

A detailed description of the mathematics behind finding the actual Greens function can also be in [20].

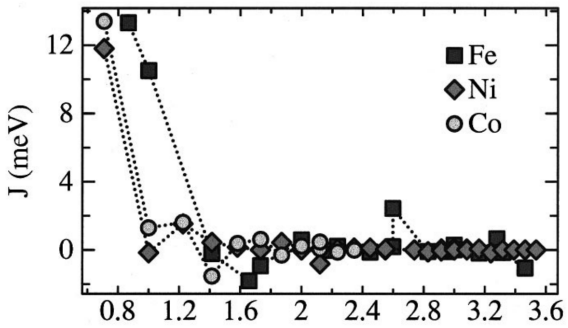


Figure 4: Exchange interactions for Nickel, Iron and Cobalt as function of distance, in units of the lattice constant (3.53 Å). **Note that the exchange parameters for nickel are scaled by a factor of 4 by the author.** [25]

2.8. Bilinear form

The bilinear form is a mapping function that maps a vector space onto a field [3], and is linear in each component [22]. It is discussed, as the model represented in the next chapter will be of this form.
In pure mathematical terms, a bilinear form $B : V \times V \to K$ has to obey the following conditions:

$$\begin{aligned} B(\mathbf{u}+\mathbf{v},\mathbf{w}) &= B(\mathbf{u},\mathbf{w}) + B(\mathbf{v},\mathbf{w}) \text{ and } B(\lambda\mathbf{u},\mathbf{v}) = \lambda B(\mathbf{u},\mathbf{v}) \\ B(\mathbf{u},\mathbf{v}+\mathbf{w}) &= B(\mathbf{u},\mathbf{v}) + B(\mathbf{u},\mathbf{w}) \text{ and } B(\mathbf{u},\lambda\mathbf{v}) = \lambda B(\mathbf{u},\mathbf{v}) \end{aligned} \tag{19}$$

Where $\mathbf{u}, \mathbf{v}$ and $\mathbf{w} \in V$. One example of a bilinear form is the dot product in $\mathbb{R}^n$. The bilinear form is a way to characterize a nonlinear optimization problem. Every bilinear form can be rewritten into a quadratic form, which is a form for which other solving methods are available, as it requires more symmetry from B. Oftentimes, the bilinear form can be written as a matrix product (20).

$$B(\mathbf{v},\mathbf{w}) = \mathbf{v}^T A \mathbf{w} \tag{20}$$

To then obtain the quadratic form, one must, in the general case, extend the matrix A to make it symmetric, as is required for a quadratic form. This means, that it is not always clear if this method, as it enlarges the matrix utilized, will help in the computation of the problem [23].

2.9. Materials

For this project, a number of commonly known materials were utilized to test the model. Below is some information about the listed materials.

2.9.1. Ni

Nickel, with atomic number 28, is a common transition metal that is mostly found in the Earths outer and inner core. In this project, we will study nickel with a face centered cubic structure with a lattice parameter of 3.53 Å. This configuration of nickel is ferromagnetic [24], and its exchange interactions can be seen in figure 4. It is studied in this project as it is one of the most simple cases of a ferromagnet.

2.9.2. MnO

[3] A field is a space where basic addition, subtraction, multiplication and division can take place as defined on $\mathbb{Q}$ [21, pp. 13–21] by $V \times V \to K$, For the purposes of this project, the field considered is $\mathbb{R}$.

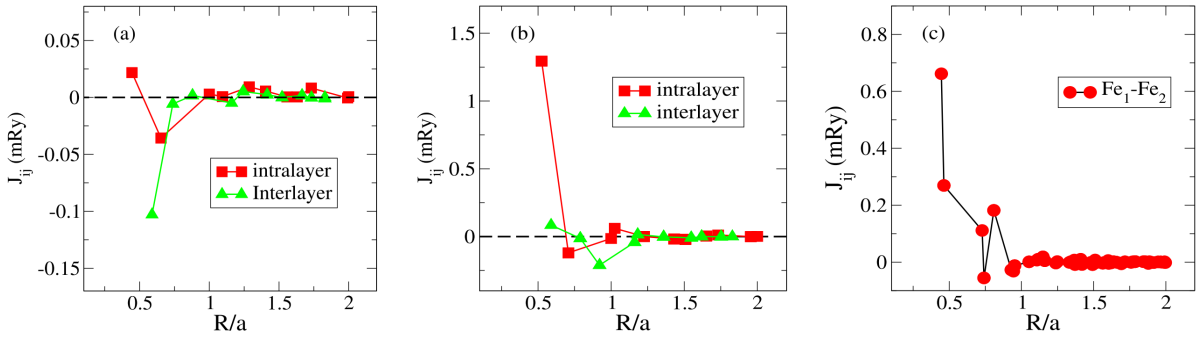


Figure 5: Exchange interactions for Fe_2P. The first, second and third plots show the Fe_1-Fe_1, Fe_2-Fe_2 and Fe_1-Fe_2 interactions, respectively [29]. The x axis shows the relative distance, divided by the lattice constant 5.813 Å. Fe_1 and Fe_2 are located at the $3f$ and $3g$ site, respectively.

Manganese oxide is a chemical compound that is utilized for numerous applications across different industries. It is antiferromagnetic in the $[\frac{1}{2},\frac{1}{2},\frac{1}{2}]$ plane [27], and has a NaCl like face centered cubic structure [28]. Its exchange parameters are given in table 1. Note that the values of J_1 and J_2 in literature do have slight variations, depending on the source [26].

	MnO	
n	E_n^{ex}	J_n
1	1.20	−30.3
2	1.18	−29.8
3	0.0	0
4	0.03	−0.8
5	0.02	−0.5
6	−0.01	0.3
7	−0.01	0.3

Table 1: The exchange parameters J_N (in K $\approx$ 0.0862 meV) and exchange energy E_n^{ex} (in mRy) per different interaction. n indicates the n-th neighbor interaction. From source [26].

2.9.3. Fe_2P

Iron phosphide is a semiconductor used in numerous fields. A lot is known about its magnetical properties, as it has recently been discovered as a potentially great material for magnetocaloric applications [29]. It has a hexagonal structure [30], with Fe atoms at the $3f$ (0.257, 0, 0) and $3g$ (0.5915, 0, 0.5) positions (expressed in the lattice vectors). Its exchange interactions according to previous work are shown in figure 5.

2.9.4. Mn_2Sb

Mn_2Sb is a little less commonly used than the previously discussed materials. It has however been studied for its interesting magnetic phase transition (from antiferromagnetic to ferrimagnetic). Recently, it (together with Mn_2Sb-based alloys) has been studied at the TU Delft for its magnetocaloric properties [31]. It is a tetragonal type structure [32], with lattice constants 3.928 Å, 3.928 Å and 6.426 Å for a, b and c respectively. The Mn atoms are located in the (0, 0, 0) and (0, 0.5, 0.7098) positions (in reference to the lattice vectors), and the Sb atom is located in the (0, 0.5, 0.2849) position. It is deemed a ferrimagnetic material in its ground state [33], with a magnetic propagation vector in the c direction[4]. The first order exchange interactions between the two different Mn sites, is determined to be -72.07 K experimentally [34], which was determined through the Ising model.

[4]A propagation vector in this sense means that in the c direction, the field is constant when traversing through the material

3. Method

3.1. Motivation

While the magnetocaloric effect (MCE) has no direct impact on model; the whole reason the model is put together ties into the MCE. As the research regarding the MCE is very much in the state of finding the optimal material for the listed applications, right now, researchers are sifting through known materials in search of a material that has a large MCE, at the right temperature, is cheap, readily available and that have a small environmental impact [5]. This means that, to find potential materials, one has to go through huge databases of materials and potentially calculate a gigantic amount of data. This model is proposed to alleviate some of the work that goes into this research, as the model simplifies the routine of finding the magnetic ordering[5] of materials.

3.2. Data acquisition

For the purpose of testing the validity of the model, four different well known materials were chosen to check the workings of the model. For these, we obtain the desired properties via a multi step process. The most important property for the model, the exchange parameters, are acquired as follows. First, the Vienna ab-initio simulation package (VASP) is used to approximate the density of states and bloch wave solution of the material. Then, Wannier90 [35] is used to calculate wannier functions from the VASP output. Then as a final step, TB2J [36] is used to calculate the exchange parameters of the material, using the output of Wannier90. The final exchange parameters are then compared to literature.

3.2.1. Vienna ab-initio simulation package

There are multiple implementations of DFT. The one utilized in this study is the Vienna ab-initio simulation package (VASP) [37, 38, 39, 40]. VASP is a well known, versatile program used to model materials on an atomic scale. It does this via the only possible way, by using the DFT method. How VASP functions exactly is beyond the scope of this project.

3.2.2. Wannier90

Wannier90 is a program that rewrites numerical Bloch functions into maximally localized Wannier functions (see section 2.6), which span the same space as the initial Bloch functions. It does so by minimizing a functional that represents the total spread of the Wannier functions in real space. This minimization is carried out directly from the Bloch functions represented in reciprocal space, and itself takes place in a space of unitary matrices[6] that describe the rotation around each k point. The exact implementation of Wannier90 goes beyond the scope of this work. Important to discuss however, is on what Wannier90 relies to obtain accurate Wannier functions from the given input. Next to inputing the Bloch functions obtained from VASP, the user is also asked to specify which ortibals from which atoms are to be taken into account. Next to that, the user also inputs a frozen window: i.e. a window in which all the orbitals specified are to be found. Generally; it is better for this window to be as large as possible. Therefor, an accurate way of presenting the density of states per atom is advised. The choice of the frozen window impacts the accuracy of the output of Wannier90, and should hence be taken seriously.

[5] The magnetic ordering is of importance for the search of materials, as it can tell a lot about the size of the magnetic response to an external field, its own magnetic moment and hence also about the size of the magnetocaloric effect, if the different structures for different temperatures are compared.

[6] A unitary matrix is a square (complex) matrix A for which its conjugate transpose is also its inverse, and $AA^{-1} = I$. The space of unitary matrices is defined by the matrix multiplication. [21, pp. 93–113]

3.2.3. TB2J

TB2J is a relatively new library that was developed to calculate exchange interactions, given results from a DFT calculation. It uses the green's function method to arrive at its result [36]. While TB2J does not directly support the output of VASP, it does support the output of Wannier90, hence TB2J is the last link in the data acquisition part of the project. It depends largely on the quality of the Wannier90 output to arrive at accurate results. This means that, as stated previously, the Wannier90 step is crucial for the succes of a run.

3.3. The model

Mainly, the goal is to solve equation (3) for a given material. To do this properly, a base model is created, to which boundary conditions and a symmetry analyzing algorithm are added to improve precision. Per step, assumptions are made to simplify computation. Firstly, the interior point optimization method used in this model is explained. Next, the base model is introduced, then extensions are discussed one by one. The model was built in python 3.7.11.

3.3.1. Interior Point Optimization

The model uses the Interior Point Optimization (IPOPT) [41], as its optimization program. IPOPT is an open source optimization routine for nonlinear problems. It compares in performance to industry standards (for instance, KNITRO, Lancelot and LOQO) [42], and has a user friendly implementation in python for large scale problems, called GEKKO [43]. First, the general workings of IPOPT will be discussed, after which GEKKO and its underlying library APMonitor will be presented.

Working of IPOPT Firstly, a short word on the workings of IPOPT. IPOPT is a deterministic optimization routine for nonlinear problems. It guarantees a global minimum when the problem is convex, and if the problem is concave, it can arrive at a local minima. Given a problem (for instance the one described in equation (21)), it first writes it into standard form[7], and then introduces a barrier function for all variables in the following manner:

$$\begin{array}{lll} \min & f(\mathbf{x}) & \\ \text{s.t.} & c(\mathbf{x}) = 0 & \\ & x_i \geq 0, \quad i = 1, \ldots, n & \end{array} \tag{21}$$

$$\begin{array}{lll} \min & f(\mathbf{x}) + \mu \sum_{i=1}^{n} \ln x_i & \\ & c(\mathbf{x}) = 0 & \end{array} \tag{22}$$

To get rid of the last inequality constraints. Here, $f(x_i)$ is the objective function, x_i are the variables and μ is a positive constant that is lowered per iteration step. $c(x_i)$ are the (potentially nonlinear) constraints. The barrier function $\mu \ln x$ functions exactly as the constraint in (21) would for small values of μ, as it runs into an asymtote at $x = 0$. Therefore, this is a valid simplification of the given model, and one that is central to the functioning of IPOPT. The IPOPT method does not solve the presented problem in (22) directly, but instead evaluates when its derivative is equal to zero, in a clever way. For further details, we refer to [41].

GEKKO GEKKO is a python package that handles machine learning and optimization. In GEKKO the user can build and run models using a couple of freely available solvers, for a number of different

[7] Standard form in optimization means that there are no inequality constraints, and all variables are ≥ 0. this is done by introducing slack variables to an equation, for example $f(x) \leq b$ will become $f(x) + s = b$, where $s \geq 0$ is the slack variable.

problems. It utilizes the Advanced Process Monitor [44] package to actually work with the models provided by the user. APmonitor is an online solving method, and GEKKO offers the functionality of APmonitor for local solving also.

3.3.2. Assumptions

The overall model makes a couple of important assumptions. To make matters more clear, an overview is shortly listed in this section.

- Magnetic moments of atoms are considered to be one dimensional.
- Magnetocristalline anisotropy is not taken into account.
- Materials are only checked for constant or alternating behaviour of moment configurations between unit cells.
- The ideal configuration is assumed to be magnetically symmetric to the same order in all three directions.
- We assume that the unit of repetition is smaller or equal to the used supercell, for each lattice direction.

3.3.3. The basic model

The basic model works as follows. The user inputs a supercell size (i.e. the amount of cells in each lattice direction), the material structural information and the exchange parameters of the material. The exchange parameters and structural information have to, as of the current version, be given in the form of TB2J (or RSPt[8]) output and VASP format, respectively. The program then uses this information to construct a minimization model that obeys the following equations (23):

$$\begin{aligned} \text{min} \quad & -2 \cdot \sum_{i=1}^{n} \sum_{j<i} J_{ij} s_i s_j \\ \text{s.t.} \quad & |s_i| \leq |d_i|, \quad i = 1, \ldots, n \end{aligned} \tag{23}$$

Where J_{ij} is the exchange interaction between the two atoms, s_i and s_j are the magnetic moments of atom i and j, respectively. d_i is the maximum magnetic moment of atom i, obtained from the input TB2J file. As d_i can be negative, the absolute value is used. The atoms, while of course present in a 3D grid, are each given an index $\in \{1, 2, \ldots, n\}$, to make the summation in (23) easier. This does not affect other computations. The summation makes sure to sum over each interaction exactly once. Note that the term $J_{ii} s_i s_i$ is disregarded $\forall i \in \{1, \ldots, n\}$, as an atom does not have an exchange interaction with itself. Furthermore, the magnetocrystalline anisotropy is disregarded, and only spins in one direction are considered. For our purposes, this one dimensional moment is a logical assumption, as equation 23 is minimized for vectors for which either their dot product is maximal or minimal. Therefore the simplification to one dimension makes sense, as for vectors with the given bounds, this is the case when $\mathbf{s_i} = \mathbf{s_j}$ or $\mathbf{s_i} = -\mathbf{s_j}$. Magnetocrystalline anisotropy is disregarded (see 2.4).
To start the model, GEKKO requires an initialization from which to iterate from. One of three initialization methods (24, 25, 26) is used.

$$|s_i| < |d_i| \tag{24}$$

$$0 < s_i < |d_i| \tag{25}$$

$$0 < m_i s_i < |d_i| \tag{26}$$

[8] RSPt is another method for calculating electronic structures of materials, and can also be used to find exchange interactions.

Here b_i is the bound for the magnetic moment of s_i, obtained from the TB2J file[9], and m_i is the sign of b_i. Note that b_i depends solely on the position of atom i in its unit cell. Each position has a distinct bound.
The major flaw of the basic model is that it does not take into account any form of boundary conditions. This can quickly lead to errors, especially for small unit cells. This is because, for atoms that are located near the edge of the supercell, not all exchange interactions may be taken into account (as some fall outside of the supercell). As the main goal is to find a symmetric solution; the next step in the model is to impose boundary conditions.

3.3.4. Boundary conditions

As stated previously, the major flaw of the basic model is that it does not take into account any form of boundary conditions. This can quickly lead to sub optimal configurations, as not all relevant exchange interactions are taken into account (see figure 6). The main focus of the boundary conditions is to reduce the amount of unit cells required to get an accurate structure from the model.
The implementation of the boundary conditions is as follows.

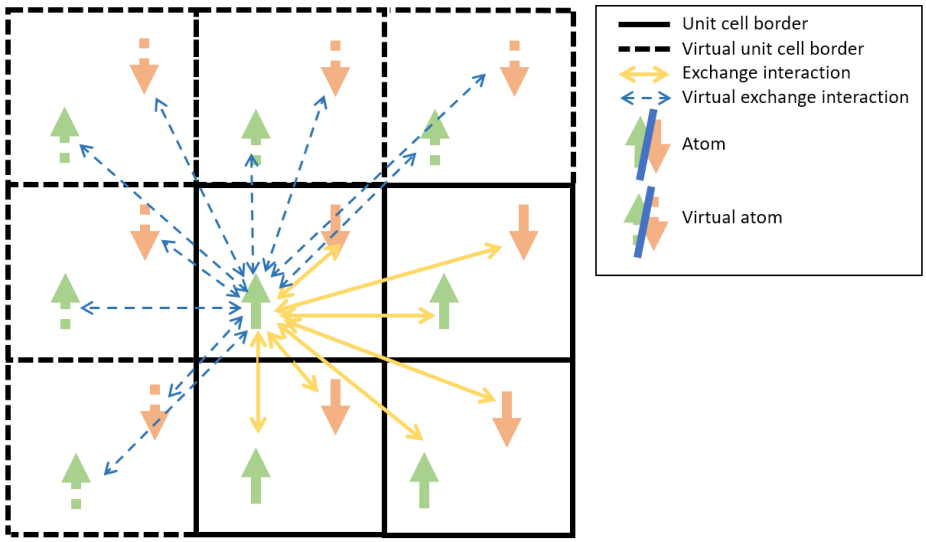


Figure 6: 2D representation of boundary conditions for one atom in a 2×2 supercell with 2 atoms per unit cell (own work).

First a set A is created that contains all atoms at the outermost unit cells of the supercell. Then, each atom $j \in A$ is moved to its equivalent positions at the opposite side(s) (what this means exactly is explained in the next paragraph) of the supercell, and the exchange interactions (between target atoms and the new position of j) are used to add the missing interactions to the objective function. A graphical representation of this procedure given in figure 7.
The choice of which equivalent positions are used for j and which exchange interactions are considered per position is made to assure no double counting occurs. For each atom $j \in A$, the following set of rules is used.
Atom j has three attributed vectors that correspond to those of its unit cell; c_1, c_2 and c_3. The index of the atom in the unit cell o is also attributed. With these four parameters, each different atom is distinguishable. For each of the vectors c_1, c_3 and c_3 it is checked if belongs to the outer most unit cells in its direction. Hereby it is determined if j belongs to an edge, face or corner of the supercell. For each of the three cases, the added exchange interactions are given in equations 31, 32 and 33 respectively.

[9]If the exchange interactions were provided via RSPt, the moments of atoms have to be provided separately.

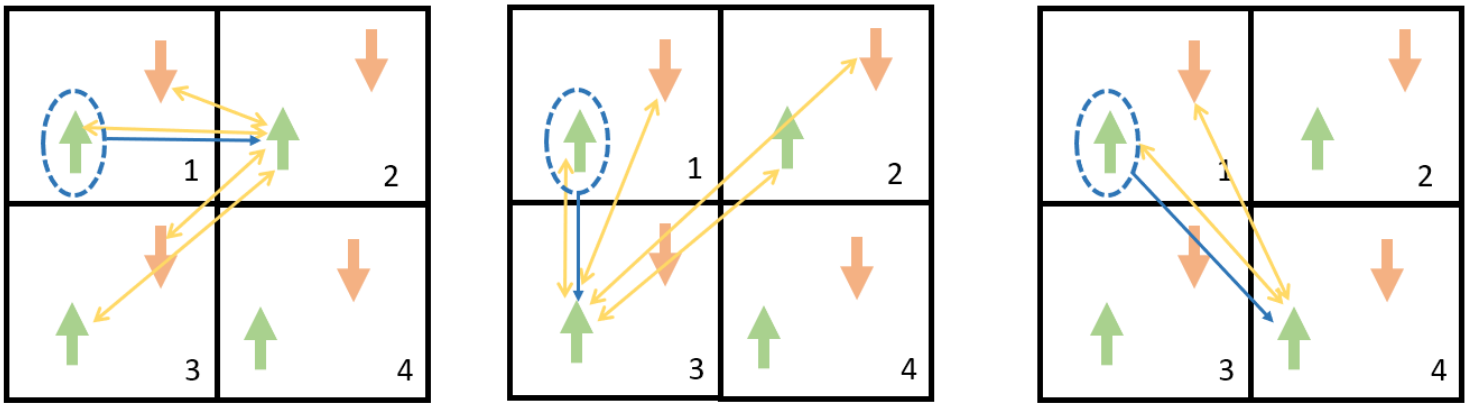


Figure 7: Graphical representation of the boundary condition implementation for one atom in a 2×2 supercell. The circled atom is virtually moved to its equivalent location in an other unit cell, shown by the blue arrow. Then, selected exchange interactions are added between the original atom and the target atoms. Hereby the virtual cells shown in figure 6 are added to the system (own work).

Let $(b_1, b_2, b_3, b_4, b_5, b_6)$ be the set of lower and upper bounds of the supercell (with $b_l \in \mathbb{Z}$ for $l \in \{1, \ldots, 6\}$), with (b_1, b_2, b_3) lower bounds and (b_4, b_5, b_6) upper bounds for the a, b and c lattice vectors respectively. B, M and B_l for $l \in \{1, 2, 3\}$ are defined as follows (27).

$$\begin{aligned} B &= \{(a_1, a_2, a_3, n) : a_1 \in B_1, a_2 \in B_2, a_3 \in B_3, n \in M\} \\ M &= \{n : n \text{ is a unique magnetic atom, per unit cell}\} \\ B_l &= \{b_l, b_l + 1, \ldots, b_{l+3} - 1, b_{l+3}\} \qquad \text{for} \quad l \in \{1, 2, 3\} \end{aligned} \tag{27}$$

Given these definitions and the location of atom j, sets $C_j(l), D_j(m)$ and E_j are defined as follows $\forall j \in A$ (28, 29, 30).

$$C_j(l) = B \setminus \{(a_1, a_2, a_3, n) : [n \in M, a_l \in B_l \setminus \{c_l\}, a_m \in B_m \text{ for } m \in \{1, 2, 3\} \setminus \{l\}]\} \tag{28}$$

$$D_j(m) = B \setminus \{(a_1, a_2, a_3, n) : [n \in M, a_l \in B_l \setminus \{c_l\}, a_m \in B_m \text{ for } l \in \{1, 2, 3\} \setminus \{m\}]\} \tag{29}$$

$$E_j = B \setminus \{(a_1, a_2, a_3, n) : n \in M, a_l \in B_l \setminus \{c_l\} \text{ for } l \in \{1, 2, 3\}\} \tag{30}$$

Graphically, $C_j(l), D_j(m)$ and E_j all describe the supercell with the one, two or three faces that contain j cut away. With these sets, given the three options for atom j (face, edge or corner), we can construct the following boundary conditions (31, 32, 33). In all three cases; $l \in \{p : c_p = b_p \vee c_p = p_{l+3}\}$ and m = $\{1, 2, 3\} \setminus \{l\}$. In words, l describes the lattice coordinates of j where c_p is at a max or minimum, and m describes the ones where j is not.

$$f_F(j) = -2\dot{\sum}_{i \in C_j(l)} J_{iq} s_i s_j \tag{31}$$

For j at a face where γ (with lattice vectors and atom index $\gamma_1, \gamma_2, \gamma_3, n$) is the equivalent atom at the opposite face (for example $\gamma_1, \gamma_2, n = c_1, c_2, n$ respectively and $\gamma_3 = b_6$ when j is at the minimum

bound of the c vector in the supercell). Note that here, l is equal to the coordinate where j is at the face and hence only has one value. Only one equivalent position of j is used.

$$f_E(j) = -2 \cdot \left(\sum_{l} \sum_{i \in C_j(l)} J_{iq} s_i s_j + \sum_{i \in D_j(m)} J_{ir} s_i s_j \right) \tag{32}$$

For j at an edge. Here, q is the atom that is at the edge opposite to j. Note that here, l has two possible values, and q has one value. Hence a total of three equivalent positions of j are used.

$$f_C(j) = -2 \cdot \left(\sum_{l} \sum_{i \in C_j(l)} J_{iq} s_i s_j + \sum_{m} \sum_{i \in D_j(m)} J_{ir} s_i s_j + \sum_{i \in E_j} J_{it} s_i s_j \right) \tag{33}$$

For j at a corner. Here, t is the atom that is the equivalent atom at the corner opposite to j. In the corner case, l and q have three possible values, and t has one. Hence, a total of 7 equivalent positions of j are used.
The last step is to rewrite the model (23) in such a way that it includes boundary conditions. Define:

$$\begin{aligned} A_F &= \{a \in A : a \text{ is at a face}\} \\ A_E &= \{a \in A : a \text{ is at an edge}\} \\ A_C &= \{a \in A : a \text{ is at an corner}\} \end{aligned} \tag{34}$$

Where A is the set of atoms in the outermost unit cells. Then the model can be written in the following form (35):

$$\begin{aligned} \text{min} \quad & -2 \cdot \sum_{i=1}^{n} \sum_{j<i} [J_{ij} s_i s_j] + \sum_{l \in A_F} f_F(l) + \sum_{l \in A_E} f_E(l) + \sum_{l \in A_C} f_C(l) \\ \text{s.t.} \quad & |s_i| \leq |d_i|, \qquad i = 1, \ldots, n \end{aligned} \tag{35}$$

Where all terms are as previously defined. A visualization of the exact boundary condition method in 3D is presented in appendix B.
The imposed boundary conditions add a large number of equations to the model (thereby increasing computation time), but also theoretically nets better results. The exact impact will be discussed in the results section.

3.3.5. Symmetry finding algorithm

Given the model including the described boundary conditions, another addition is made to make sure that the optimal solution is found, as the interior point optimization method can get stuck at a local minimum. To account for this symptom, an algorithm is implemented that, after an initial cycle of IPOPT using the model is complete, checks if the solution is properly symmetric, and if not, reruns the model with a new initialization chosen based on the previous result. A detailed flowchart for the algorithm can be found in appendix B. In this section the algorithm is discussed in detail. First, the previous solution is analyzed. From there, a new initialization is set up, after which the new initialization is used to run the program again. Lastly, the best solution is found among all the runs.

Solution analysis During the analysis of the first run, a couple of simplifications are made. Firstly, the algorithm only considers the sign of each magnetic moment.
The spins, as all are in one dimension, are now either +1 or -1. With this knowledge, the program now checks for a unique configurations of spins in a unit cell. With this check; the inverse of a configuration is

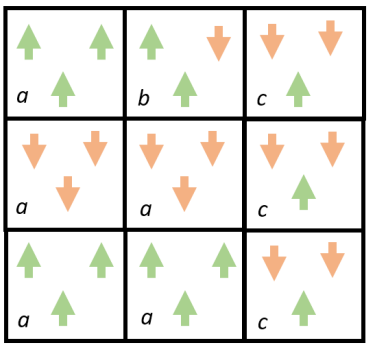


Figure 8: An example 2D structure. Different configurations are labeled in a, b and c. As this configuration has 9 cells, both configurations a and c are considered prominent.

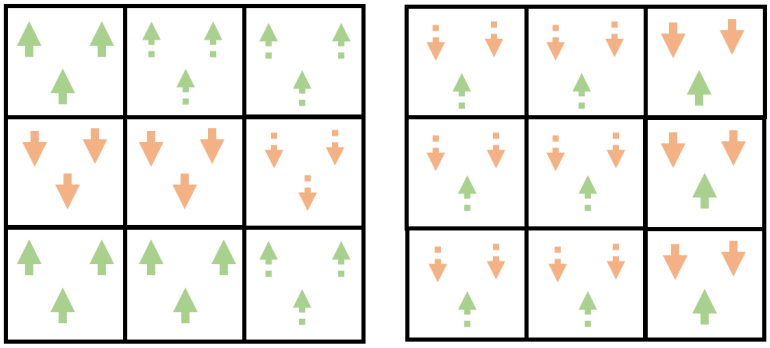


Figure 9: The new starting configurations, initialized from the structure in 8. Changed atoms are highlighted by the smaller, dotted arrows to indicate their different starting size due to γ. Note that the extra completely symmetric solution step is not plotted here, though in 3D this would be considered (own work).

considered the same. See figure 8 for a 2D example. As soon as all different configurations of unit cells are determined, the program finds the ones that occur most often in the supercell. The configurations that constitute more than 30% of the total amount of unit cells in the supercell are saved. If there is no configuration that meets this quota, it will run the optimization step again, from a different random initialization, in hopes of finding a better result. With all the saved configurations, the program follows the following procedure to produce a new initialization.

Constructing new initialization(s) Assuming a prominent configuration is given, the program now checks certain symmetry conditions. In every lattice direction, the program makes an analysis of the repetition of the current selected configuration, to determine in how many unit cells the configuration repeats. Note that as per the current version, the model only checks for constant or alternating patterns. For a 2D example, see figure 9. Then, in each direction, the material is determined to be either constant, alternating, or neither of the two. The program then runs the following analysis to construct the next starting iteration.
First it checks if the material is considered perfect. This distinction is given when the material only consists of one configuration, the pattern in all three directions is the same and the material completely obeys this pattern. If a material is perfect, no second run is executed, as the current solution is considered good enough.
If the material is not perfect, a number of steps are executed to arrive at a new initialization (or initializations).

- Firstly, if the pattern is neither constant or alternating in any direction, in the new configuration it is set to constant in that direction.
- Next, a new configuration is built by pasting the new pattern onto the material. If a unit cell is altered, its new moments are also multiplied by the parameter $\gamma \in (0, 1)$, see figure 9
- If the given pattern is not the same in all three directions (so either completely constant or completely alternating), a second new configuration is built by pasting the completely constant or completely alternating pattern onto the material (constant or alternating is chosen if the majority of the pattern is constant or alternating respectively).

Running new initialization(s) As soon as all configurations are checked and their respective new initializations are built, one by one, new runs are executed given these initializations.

4. Results

4.1. Exchange interactions

For Ni and MnO, the exchange interactions were calculated. For Fe_2P and Mn_2Sb, the literary values for the exchange parameters were used. In this section, the results for the exchange interactions for nickel and manganese oxide will be reported.

4.1.1. Ni

For nickel, the calculated exchange interactions can be found in figure 4. The corresponding moment of the atom (only one atom as the primitive cell of nickel was used) is equal to 0.5542 μ_B.

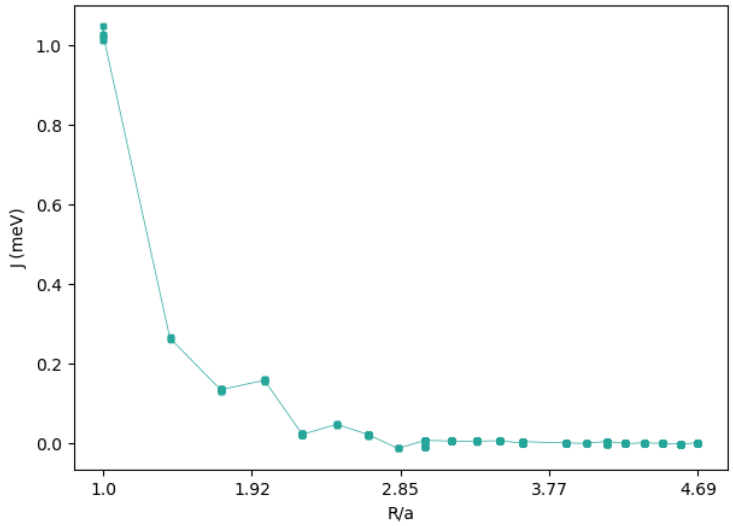


Figure 10: Exchange interactions found for Ni, plotted against distance divided by the lattice parameter (= 3.53 Å).

4.1.2. MnO

The exchange interactions calculated for MnO can be seen in figure 11. The moments for the Mn atoms located at (0,0,0), (0,$\frac{1}{2}$,$\frac{1}{2}$), ($\frac{1}{2}$,0,$\frac{1}{2}$) and ($\frac{1}{2}$,$\frac{1}{2}$,0) are equal to 2.8643, -2.8768, -2.8399 and 2.8797 μ_B respectively.

4.2. Model results

In this section, the results of the model will be discussed. First, the basic model (without the inclusion of the boundary conditions) and its outcomes will be discussed for all used materials. This will include the optimal structure found per material, the percentage of runs that arrived at this structure and the average time of a run. All quantities listed, will also be reported for each different initialization method.
Next, the same discussion will be held for the model including boundary conditions. Thereafter, the effectiveness of the symmetry finding algorithm will be discussed. Lastly, the combined results (basic model with boundary conditions and algorithm) will be presented. For all sections, the four materials described earlier were used (Ni, MnO, Fe_2P and Mn_2Sb) in the configurations of a $2\times2\times2$ and a $3\times3\times3$ supercell. These configurations were used, as the materials discussed all have a symmetry in each direction that is smaller or equal to the $2\times2\times2$ cell. The $3\times3\times3$ cell was also studied to check the impact of a larger supercell, as well as a cell count that is not a multiple of the materials multiplicity (i.e. 3 is not divisible by 2). For each material in each configuration, 120 runs were performed per initialization, unless stated differently. If initialization types are given, they are defined as follows (see 24, 25 and 26).

$$\begin{aligned} \text{Initialization 1:} \quad & |s_i| < |d_i| \\ 2: \quad & 0 < s_i < |d_i| \\ 3: \quad & 0 < m_i s_i < |d_i| \end{aligned}$$

For all different executed tests, the general convergence of different runs (i.e. energy per iteration) are recorded in appendix C. Secondly, appendix C also contains a note on energy differences between different versions of the model, and some examples of suboptimal configurations found in the runs for MnO. These suboptimal configurations will not be commented on further, but rather serve as an example for the reader.
As a final general comment, for all computation times holds that computations were executed on a laptop with 8 GB of DDR3 RAM, with a 4-core i5-8250U Intel CPU.

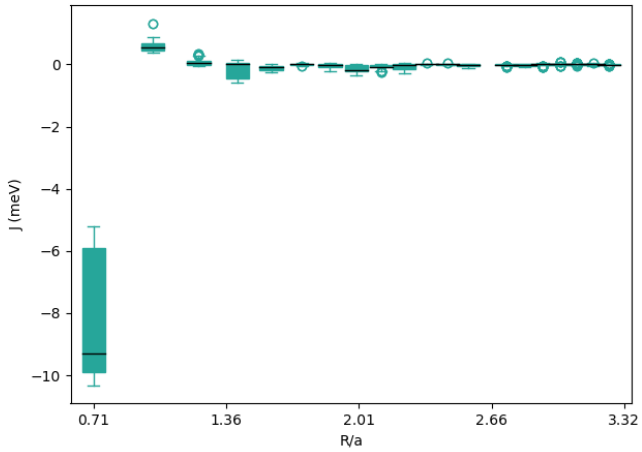


Figure 11: Exchange interactions found for MnO, plotted against distance divided by the lattice parameter (= 3.53 Å). The boxes turquoise specify the 2nd and 3rd quartile, the black lines the average and the turquoise lines the 1st and 4th quartile of the data gathered per distance. The turquoise dots represent outliers from this range.

4.2.1. Basic model

In the runs conducted for the basic model, multiple different outcomes were found. Per outcome, the resulting exchange energies were recorded, as that is in the end the value that the model minimizes. Per distinct energy (up to a certain numerical value which is smaller than 1% of the resultant energy), the observed outcome has a different configuration. All recorded energies are shown in figure 13. Interestingly, only a select amount of energy levels are reached, and most energies are never reached as a final outcome. From the runs as shown in figure 13, each setup has a clear preferred ground state energy. These configurations are given in figure 12. Note that both the 3×3×3 and 2×2×2 configurations for all materials gave the same optimal structure, even though there is a difference in the minimal energy recorded per unit cell. This difference in energy per configuration is due to a different maximum of order of exchange interactions being taken into account, and is further elaborated upon in appendix C.
For the conducted runs, the number of times that the model reaches these optimal configurations is listed in table 2. The average run times per initialization are given in table 3.

		Basic model			
	Initialization	1	2	3	Total
Material	Supercell	Successful runs		/120	/360
Ni	2×2×2	118	120	120	358
	3×3×3	116	120	120	356
MnO	2×2×2	47	65	80	192
	3×3×3	28	33	65	126
Fe_2P	2×2×2	54	103	96	253
	3×3×3	9	100	97	206
Mn_2Sb	2×2×2	101	115	108	324
	3×3×3	84	86	90	260
	Total	557	742	776	2075
	Out of	960	960	960	2880
	Percentage	58.02	77.29	80.83	72.05

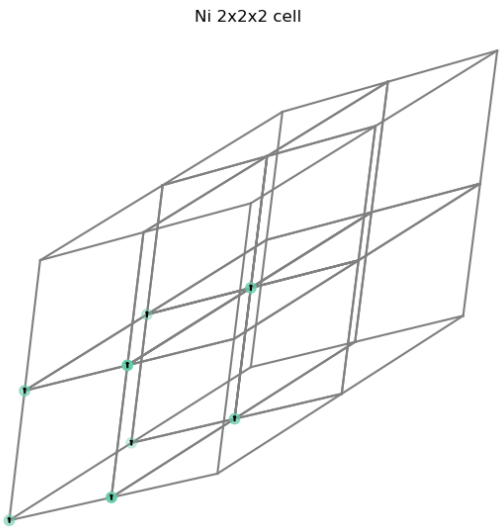
Table 2: Amount of optimal solutions for the basic model per material and per initialization. A run counts as a successful one when it reached the optimal energy/ magnetic configuration.

	Initialization	1	2	3	Total
Material	Supercell	Average time (s)			
Ni	2×2×2	0.59	0.59	0.6	0.6
	3×3×3	0.7	0.68	0.68	0.69
MnO	2×2×2	1.14	1.18	1.1	1.14
	3×3×3	6	6.73	5.25	5.99
Fe_2P	2×2×2	1.74	1.6	1.62	1.65
	3×3×3	16.59	12.32	12.59	13.83
Mn_2Sb	2×2×2	0.53	0.53	0.53	0.53
	3×3×3	3.39	3.33	3.29	3.34

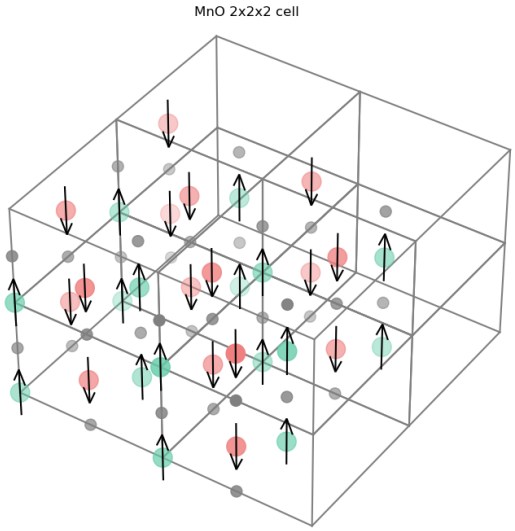
Table 3: Average run times of the model in seconds, per iteration.

In these results we can see that the accuracy of the basic model really depends on the material it is used on. Where for Ni, the model performs quite well, for MnO, the model has a way lower overall success rate. The different initializations also impact the final result, most evidently shown in MnO and Fe_2P.
The run time per material and configuration varies heavily, as shown in table 3. The largest difference in

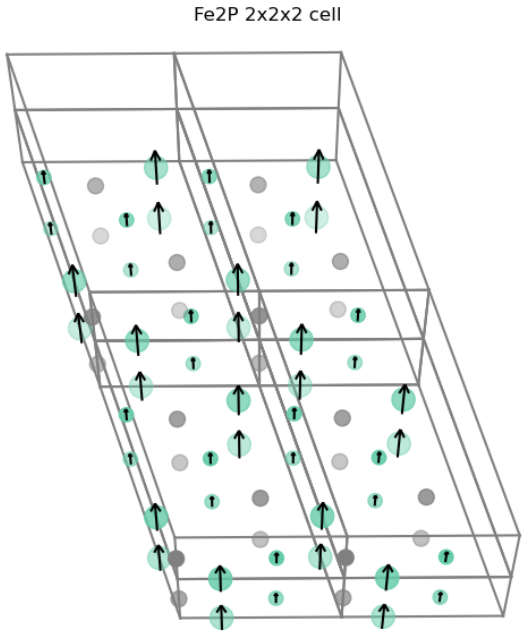
run time between initializtations is about 30%, for Fe_2P 3×3×3. From figure 13, we can see that while MnO and Fe_2P have a lot of often occurring suboptimal solutions, Mn_2Sb and Ni only have one that is really prominent.



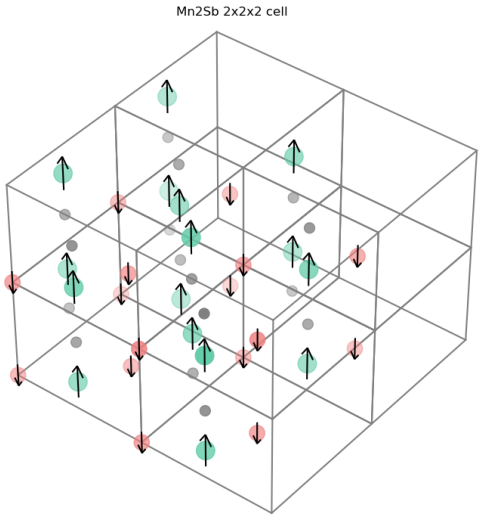
(a) The optimal Ni structure. As all calculations for nickel were done in its primitive unit cell; this is the resultant ferromagnetic configuration with moments of 0.5 μ_B.



(b) The optimal MnO structure; antiferromagnetic configuration with ferromagnetic planes in the ($\frac{1}{2}$,$\frac{1}{2}$,0) direction. Its moments are equal to 2.86 μ_B, direction is given by the arrows in the figure.



(c) The optimal Fe_2P structure according to the model. It is ferromagnetic, with the $3f$ and $3g$ positions having a magnetic moment of 0.81 μ_B and 2.1 μ_B, respectively.



(d) The optimal configuration for Mn_2Sb. The (0, 0, 0) and (0, 0.5, 0.7098) positions have a magnetic moment of 2.1 μ_B and 3.1 μ_B, respectively.

Figure 12: In all structures, magnetic atoms are shown in red or green, nonmagnetic atoms in gray. The size of the red or green dots indicates the size of the moments. For all materials the exact moments per atom vary by $\approx 0.05\%$ between unique atom positions, as they originate from a numerical input.

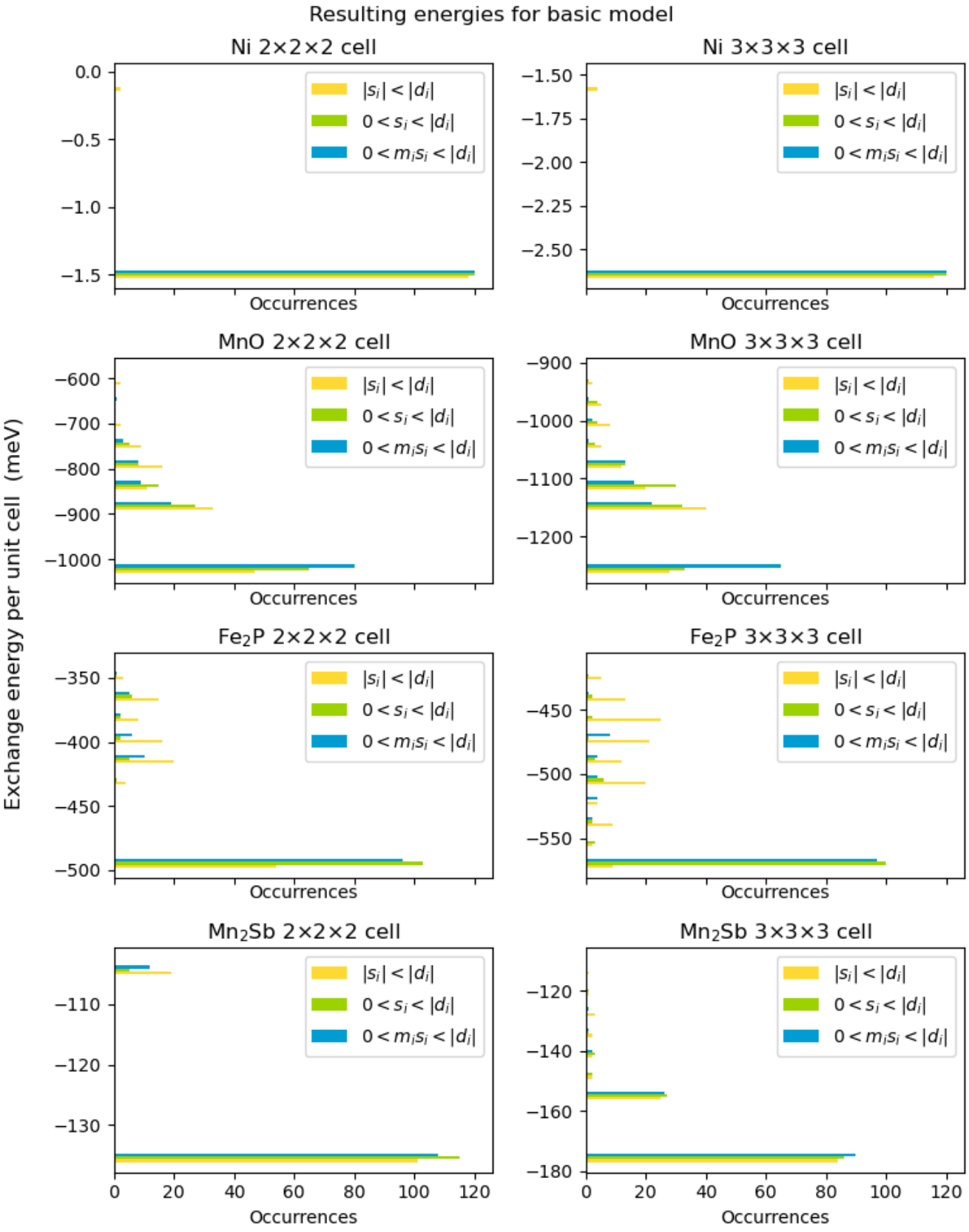


Figure 13: The full convergence results for the basic model.

4.2.2. Including boundary conditions

For the model with the inclusion of boundary conditions, the same results procedure is followed as with the base model. Across the board, the boundary conditions add more accuracy to the model, but heavily increase the computation time for more complex configurations. The full final energies attained can be found in figure 14. Due to the implementation of the boundary conditions (as more exchange interactions are evaluated), the energies of the model including boundary conditions are increased compared to the base model, still the optimal solution between the two is the same. This added energy effect is further discussed in appendix C. The accuracy and time performance of the model including boundary conditions can be found in tables 4 and 5, respectively.

		Basic model including BC's			
	Initialization	1	2	3	Total
Material	Supercell	Successful runs		/120	/360
Ni	2×2×2	105	120	120	345
	3×3×3	120	120	120	360
MnO	2×2×2	80	94	96	270
	3×3×3	68	68	94	230
Fe_2P	2×2×2	61	108	114	283
	3×3×3	25	110	112	247
Mn_2Sb	2×2×2	104	116	112	332
	3×3×3	94	95	109	298
	Total	657	831	877	2365
	Out of	960	960	960	2880
	Percentage	68.44	86.56	91.35	82.12

Table 4: Results for the basic model including boundary conditions, per initialization and per material configuration.

	Initialization	1	2	3	Total
Material	Supercell	Average time (s)			
Ni	2×2×2	0.61	0.59	0.59	0.60
	3×3×3	1.29	1.23	1.22	1.25
MnO	2×2×2	2.37	2.54	2.17	2.36
	3×3×3	24.14	25.62	20.77	23.50
Fe_2P	2×2×2	5.03	4.25	4.29	4.53
	3×3×3	72.79	52.84	52.71	59.45
Mn_2Sb	2×2×2	1.84	1.81	1.78	1.81
	3×3×3	8.29	8.23	8.14	8.22

Table 5: Run times for the model with the addition of boundary conditions per initialization and material configuration.

Mostly, the same trends that were present as in the basic model are still present in the basic model with boundary conditions. Most notable is that the model with boundary conditions is more accurate than the basic model (seen by the overal percentages in table 4). Secondly, the average run time for the model with boundary conditions is a fraction higher than the run times in the basic model. Lastly, when looking at figure 14, we can see that the suboptimal solutions are differently distributed compared to the basic model.

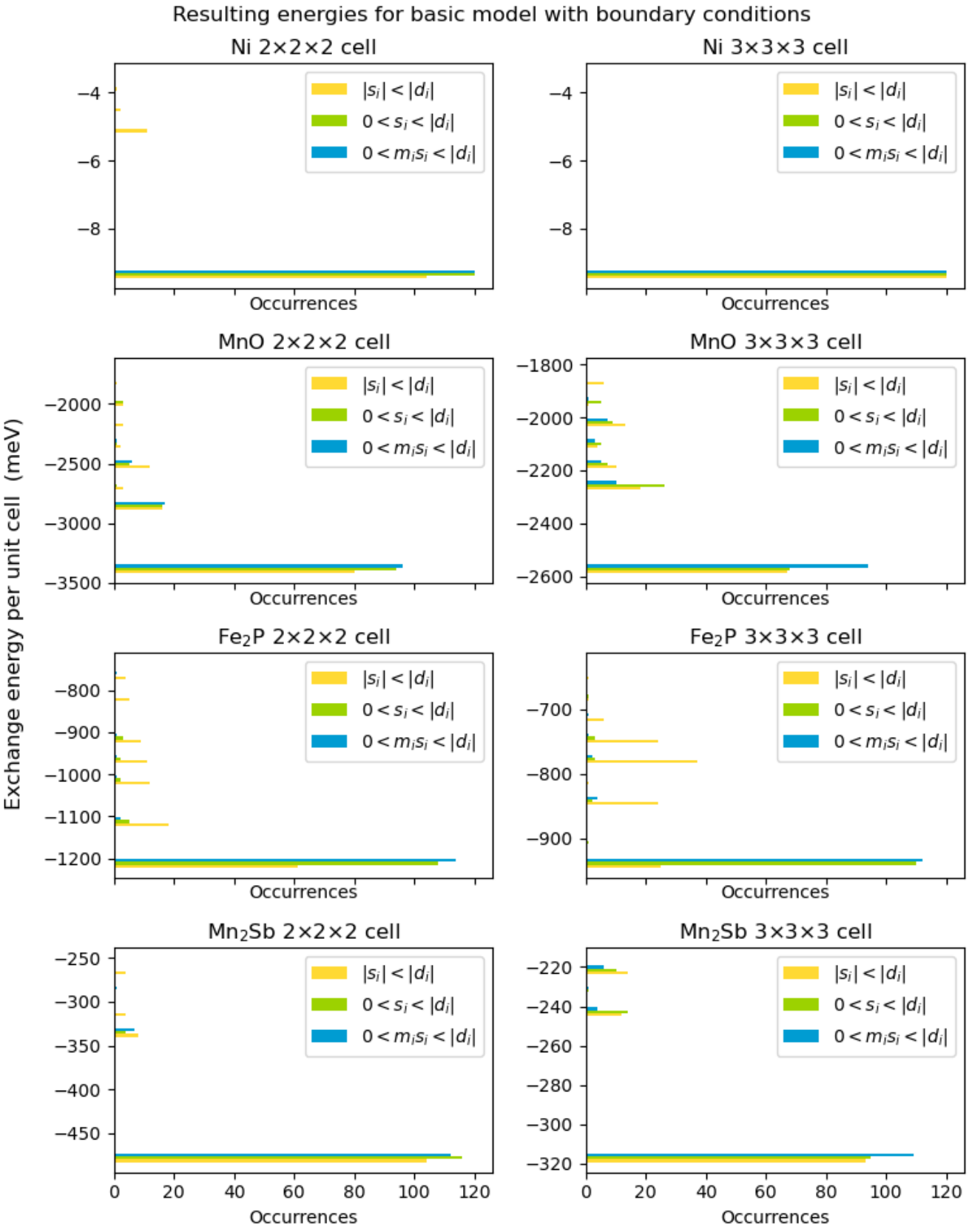


Figure 14: The full convergence results for the basic model with the inclusion of boundary conditions.

4.2.3. Symmetry finding algorithm

The algorithm will be evaluated in a different fashion than the previous two versions of the model. First and foremost, the algorithm was tested only with the model version that includes the boundary conditions. As it only kicks in when a solution is considered not 'optimal enough', a test set was constructed on which the algorithm was executed. The test set contains all sub-optimal solutions from the boundary conditions run, see figure 15. For the values of γ from 0.3 to 0.8 with increments of 0.1, the test set was executed and the success rate and running times are reported in tables 6 and 7. These values of γ were chosen as a logical spread of values between 0 and 1. The final attained energies of the runs are presented in 16. The convergence of the test set of each material configuration for different values of γ is reported in appendix C.

	γ	0.3	0.4	0.5	0.6	0.7	0.8	
Material	Supercell	Percentage of successful runs						Total runs
Ni	2×2×2	100	100	100	100	100	100	15
MnO	2×2×2	97.78	98.89	98.89	100	38.89	38.89	90
	3×3×3	100	95.38	96.15	98.46	100	98.46	130
Fe_2P	2×2×2	100	100	94.81	93.51	61.04	100	77
	3×3×3	100	100	97.35	46.9	98.23	100	113
Mn_2Sb	2×2×2	100	100	100	100	100	21.43	28
	3×3×3	98.39	72.58	72.58	98.39	100	98.39	62
Total successful		512	491	485	447	428	435	515
Total percentage		99.42	95.34	94.17	86.8	83.11	84.47	

Table 6: Percentage of successful runs of the algorithm, per value of γ. No values for Ni 3×3×3 are present, as this configuration executed perfectly for the model with boundary conditions in all 360 runs.

	γ	0.3	0.4	0.5	0.6	0.7	0.8
Material	Supercell	Average total runtime (s)					
Ni	2×2×2	1.1	1.09	1.1	1.14	1.15	1.17
MnO	2×2×2	6.08	6.01	5.95	5.99	5.84	6.2
	3×3×3	61.42	59.78	55.85	57.01	55.05	53.22
Fe_2P	2×2×2	9.46	9.33	9.28	9.34	9.26	9.12
	3×3×3	171.55	157.85	165.52	166.01	160.11	165.65
Mn_2Sb	2×2×2	3.58	3.72	3.98	3.84	3.65	3.82
	3×3×3	16.97	17.52	17.53	16.71	16.23	15.98

Table 7: Run times per value of γ, per configuration.

We can see that which value for γ is selected, quite heavily impacts the result of the run. This effect will be further discussed in the next section. Secondly, the average run time for the different values of γ differ at most around 10% of the total run time. The success rate of the algorithm on the configurations is also heavily impacted for different values of γ. Examples are Fe_2P 3×3×3, where all values of γ have a (near) 100% success rate, except for $\gamma = 0.6$ which only has one of 46.9%. The same effect seems to occur with Mn_2Sb, this time with $\gamma = 0.8$. Overall, when comparing the figure 15 and 16, we can see that the addition of the algorithm solves a lot of the previously inaccurate cases.

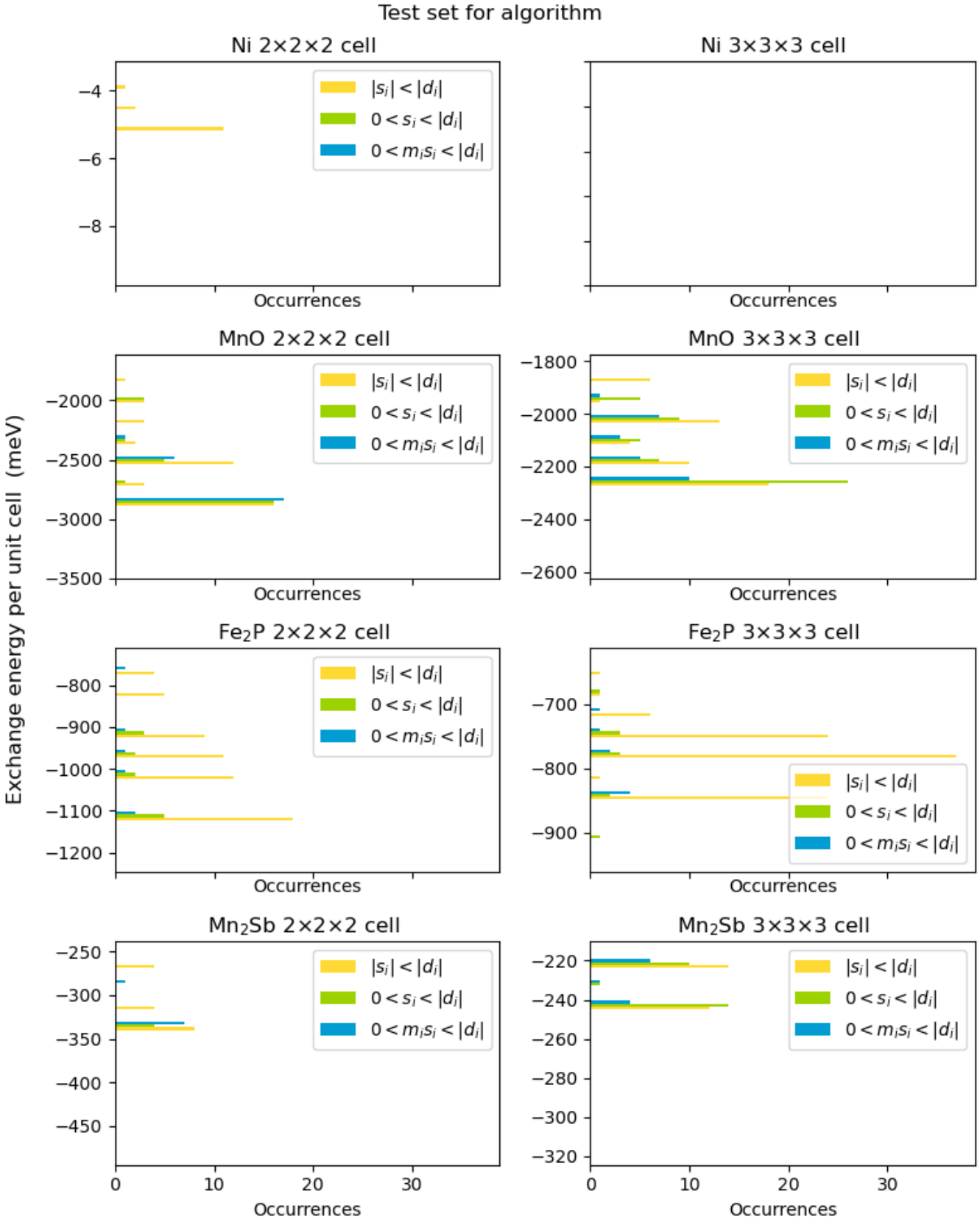


Figure 15: The full convergence results for the test set for the algorithm. This corresponds to 14, with the exclusion of the optimal results. Note that the occurrence axis is differently scaled compared to the previous figures.

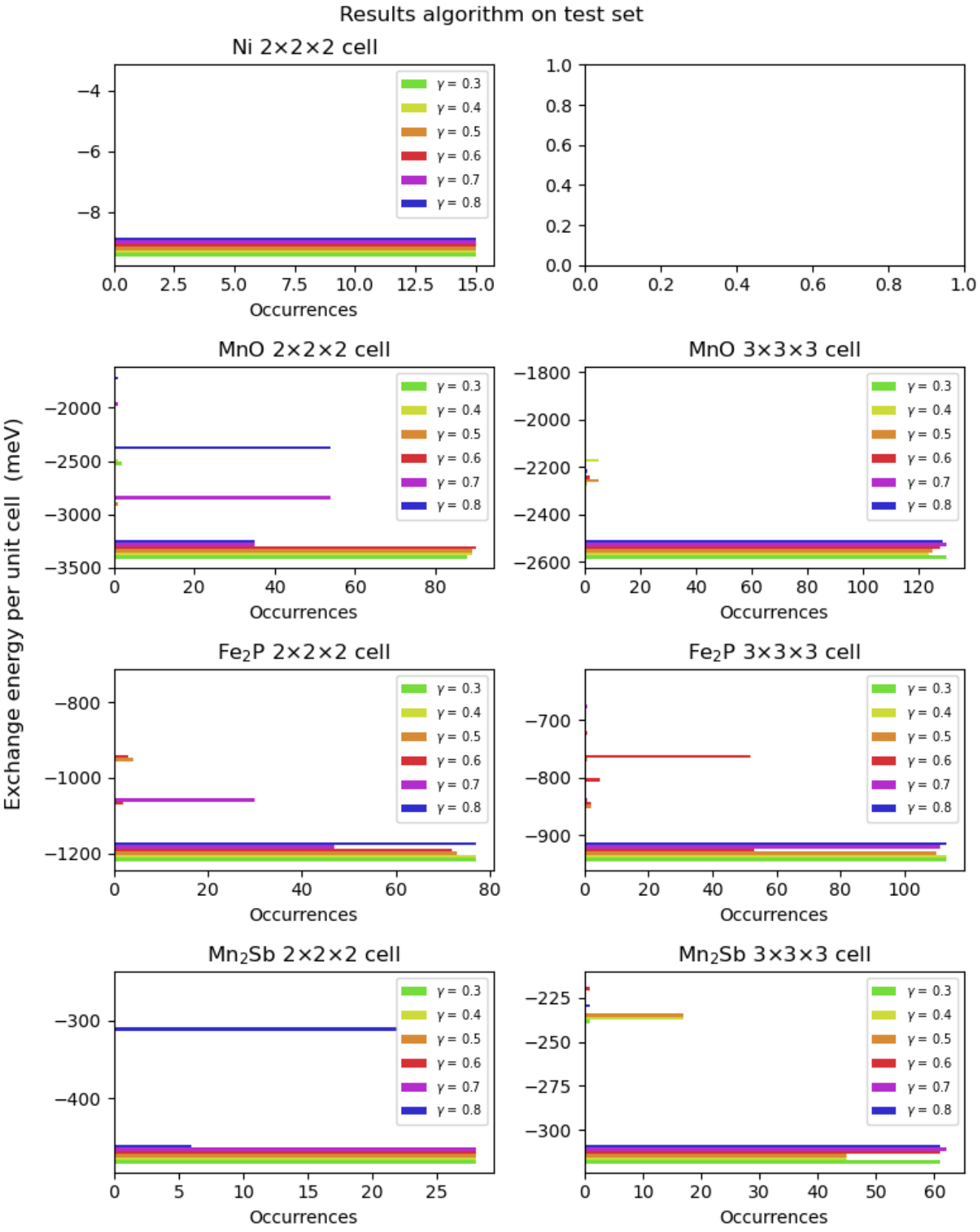


Figure 16: The full convergence results for the algorithm on the test set presented in figure 15, for different values of γ.

4.2.4. Versions comparison

Given the results mentioned above, as the algorithm was executed on the set of sub optimal solutions of the model with boundary conditions, without loss of generality, we can combine the results to virtually construct a total average run time and accuracy per value of γ on the complete set of runs from the model including boundary conditions. This is done to more accurately represent real life situations, and to be able to compare its results to those from sections 4.2.1 and 4.2.2. The motivation for why this is allowed is straightforward, as each different run of IPOPT is independent and only depends on initialization, neither data set is affected by this virtual construction. The combined results can be found in tables 8 and 9.

	γ	0.3	0.4	0.5	0.6	0.7	0.8	
Material	Supercell	Percentage of successfull runs						Total runs
Ni	2×2×2	100	100	100	100	100	100	360
	3×3×3	100	100	100	100	100	100	360
MnO	2×2×2	99.44	99.72	99.72	100	84.72	84.72	360
	3×3×3	100	98.33	98.61	99.44	100	99.44	360
Fe_2P	2×2×2	100	100	98.89	98.61	91.67	100	360
	3×3×3	100	100	99.17	83.33	99.44	100	360
Mn_2Sb	2×2×2	100	100	100	100	100	93.89	360
	3×3×3	99.72	95.28	95.28	99.72	100	99.72	360
Total successfull		2877	2856	2850	2812	2793	2800	2880
Total percentage		99.895	99.17	98.96	97.64	96.98	97.22	

Table 8: Total accuracy as projected on the model including boundary conditions data set, value of γ versus material configurations.

	γ	0.3	0.4	0.5	0.6	0.7	0.8
Material	Supercell	Average total runtime (s)					
Ni	2×2×2	0.62	0.62	0.62	0.62	0.62	0.62
	3×3×3	1.25	1.25	1.25	1.25	1.25	1.25
MnO	2×2×2	3.29	3.27	3.26	3.27	3.23	3.32
	3×3×3	37.19	36.6	35.18	35.6	34.89	34.23
Fe_2P	2×2×2	5.58	5.56	5.55	5.56	5.54	5.51
	3×3×3	94.64	90.34	92.74	92.9	91.05	92.79
Mn_2Sb	2×2×2	1.94	1.95	1.97	1.96	1.94	1.96
	3×3×3	9.73	9.82	9.82	9.68	9.6	9.56

Table 9: Average run time of performed runs in table 8. Value of γ versus material configurations.

5. Discussion

In this section, we will discuss the obtained results, what they signify, and what further research can be conducted concerning the model.
Firstly, we discuss the exchange interactions obtained for Ni and MnO. Both findings agree on the sign of J, but differ from the exchange interactions in 4 and 1 by a factor of roughly 3 (smaller) and 5 (larger) respectively. Though inaccurate, these results are in agreement enough so that they may be used for the purposes of this study.
For the optimal structures found in figure 12, when we compare the structures to the literature structures, we find that they agree with the expected structures for Ni, Fe_2P and Mn_2Sb. MnO is found to be antiferromagnetic, but in a slightly different lattice direction than what is specified in literature. Where in literature, MnO is AFM in the $[\frac{1}{2},\frac{1}{2},\frac{1}{2}]$ direction, the calculated structure is antiferromagnetic in the $[\frac{1}{2},\frac{1}{2},0]$ direction. This can likely be attributed to one of the processes in the gathering of the exchange interactions. This is assumed as firstly, the structure found by the model has a lower exchange energy than the literature configuration when it is compared in the model. Secondly, DFT is not a perfect representation of reality, as multiple assumptions are made, and some effects are disregarded, which will impact the resultant exchange interactions, which might have played a role in this project.

The model For the different versions of the model, with each extension, the accuracy is increased at the cost of also increasing run time. We will discuss the two different evaluated factors (initialization and the value of γ), then compare the different versions of the model, and lastly discuss some limitations of the results.

When comparing the different initializations, the initialization $0 < m_i s_i < |d_i|$ (initialization 3) (26) performed equally good to the other two initializations (24,25) in most, and significantly better than the others in some configurations. This is the case for both the basic model and the model with boundary conditions. Especially for Fe_2P in both configurations, initialization $|s_i| < |b_i|$ (initialization 1) performed substantially worse than the other two. For MnO $3\times3\times3$, initialization 3 performed remarkably better than the other two initializations. These two results can be explained intuitively by the final structure of the two materials. Where Fe_2P is ferromagnetic, setting initialization 1, where moments are randomly ordered, one would expect that this is further from the optimal result. For MnO, as initialization 3 already assigns signs that would match the optimal solution, it is logical that this initialization will have to perform 'less work' to arrive at the optimal solution. Therefore, an explanation for this superior performance is that initialization 3 uses more information of magnetic moments presented, which gives it an advantage. Given this information, initialization 3 is the initialization of choice as it not only has a higher average accuracy, but it also on average has a slightly lower run time than the other two initialization methods as shown in tables 5 and 3.

When comparing the different values of γ, $\gamma = 0.3$ clearly stands out as the optimal value. While it does have a slightly higher average run time; it more than makes up for it in accuracy, where it is (comparing results on the test set), 4% more effective than its closest competitor, with a staggering 99.42% vs 95.34%. As IPOPT is a deterministic routine, and the initialization is also deterministic per run (i.e. each different value of γ had the exact same input), this is an extremely accurate result. Surprisingly, there is a large difference in performance for each value of γ per material. This effect is most noticeable when comparing $\gamma = 0.8$ and $\gamma = 0.7$, as we can see that per material, the value of γ that performs better is different. This effect would be interesting to study, especially for other materials than the ones listed here, as there is no easy explanation for this behaviour.
Regarding the accuracy of the results, one does have to take into account that the test set did only contain 515 runs, which boils down to fairly low runs per material configuration. Next to that, only six values

for γ were tested.

When evaluating which model type to work with (basic, with boundary conditions or with algorithm), looking from an accuracy perspective, running the model including the boundary conditions and the algorithm is the clear winner. Even with the higher run times, the addition of the algorithm improves accuracy by a sufficiently large amount (99.895% vs 82.12% or 72.05% over all initializations), thus using it is highly recommended. The only other consideration is the model with boundary conditions using only initialization 3, which has an accuracy of 91.35%. However, to attain the same accuracy as the combined method in one run, this method would have to be run at least twice (for an accuracy of $1 - (0.0865)^2 = 99.25\%$), which is still lower than the accuracy of the model including the algorithm, and, on top of that, also in a longer average run time. Therefor, the use of the algorithm is highly recommended.

It should be noted, that while individually $\gamma = 0.3$ and the initialization $0 < m_i s_i < |d_i|$ performed best; the combination of the two factors has not been extensively tested, as the higher accuracy of initialization method also means that there were less test cases from that set that the algorithm worked on (only 83 test cases, see table 4).
A point of concern for the model is that the computation time between small and larger configurations is impacted greatly. It is therefor advised to examine this behaviour more thoroughly in future studies.

Recommendations For future work, there are a lot of directions to consider. In no specific order; some of those continuations are:

- Changing the model to allow magnetic moments to independently assume any direction, instead of all adhering to the same direction. Though most likely driving up computation time, this would generalize the model to also work on more complex systems.
- Make the algorithm check for more than 2 cell repetitions. This would require a more sophisticated approach to the implementation of the algorithm, and might be a time consuming task. The usefulness of this extension also heavily depends on weather or not more complex magnetic structures are of interest to the application.
- More extensive research can be done on the exact choice of the γ parameter, and the model in general. As the ideal value of 0.3 works for all test materials presented in this study, however more values of γ could be checked against more materials to find if a more general global optimal value exists.
- As the results presented in this work are promising, we are exited to see how the model, that seems quite robust for the test cases, handles a larger sample size and a larger selection of materials. As the direction of antiferromagnetism in MnO was wrongly predicted, inputting more antiferromagnetic materials would be of interest to verify the accuracy of the model on antiferromagnetic materials.
- Concerning the algorithm, an extension that smartly determines when unit cells are similar enough that they can be considered equal may be of interest. Currently, the algorithm treats each unique configuration of a unit cell as unique, no matter how close two configurations might be. For large unit cells, this might not be optimal. This extension could be useful if the model turns out to struggle with materials with large unit cells.
- Another extension that could help the algorithm, is an immediate output of the final result is presented to either itself or the user, to check if the solution actually makes sense and is not a rare error. This could also be fixed by running the model multiple times, to achieve a near 100% accurate solution.

- And lastly, one could also check how the model reacts to the slight alteration of the input conditions (i.e. different lattice parameters) to check the magnetic response of the system, and if this is also accurately predicted by the model.

6. Conclusion

In this project, a model is presented that predicts the magnetic structures of materials using exchange interactions. Boundary conditions and a symmetry finding algorithm are added to a basic model to drastically increase its accuracy to 99,895% over 2880 runs. Two factors were tested, different initialization methods and the algorithm mixing parameter $\gamma \in (0, 1)$. The latter was tested to be optimal for γ = 0.3, though only six different values for γ were tested. Interestingly, different values of γ performed rather differently per material, no general trend between the four materials was observed. The initialization method $0 < m_i s_i < |d_i|$ performed best out of the three initialization methods tested. This is likely due to it using more structural information, namely the direction of the magnetic moments of the material. The combination of using the two best performing methods was not tested. Individually however, the two performed distinctly better than their competitors, and are therefor advised to be used when using the model.

For three out of four materials tested, the optimal configuration aligned with the one shown in literature. For MnO, the result is antiferromagnetic in a different lattice direction, due to some process present in the collection of the input data.

From these findings, the model seems like a robust tool to quickly predict the magnetic structure of materials with a not too complicated structure. More testing of the model for a wider range of materials is recommended, to ensure that it functions as expected. There are also various ways to extend and to improve the model (the most prominent of which is the generalization to allow the magnetic moments to be oriented in 2 or 3 dimensions) to make it able to handle more complex systems.

References

[1] B. D. Cullity and C. D. Graham. *Introduction to magnetic materials*. Ed. by Abari R. et al. second. Vol. 1. John Wiley & Sons, Inc., Dec. 2008.

[2] M. D. E. De Baar. *Potential of the ternary Fe-Mn-Y system as cooling material for room-temperature refrigeration*. 2018.

[3] Ekkes Brück. "Developments in magnetocaloric refrigeration". In: *Journal of Physics D: Applied Physics* 38 (23 Dec. 2005).

[4] Andrej Kitanovski. "Energy Applications of Magnetocaloric Materials". In: *Advanced Energy Materials* 10 (10 Mar. 2020).

[5] Nikolai A. Zarkevich and Vladimir I. Zverev. "Viable materials with a giant magnetocaloric effect". In: *Crystals* 10 (9 Sept. 2020), pp. 1–28.

[6] David J. Griffiths and Darrell F. Schroeter. *Introduction to Quantum Mechanics*. Cambridge University Press, Aug. 2018. ISBN: 9781316995433.

[7] Xueyang Li et al. "Spin hamiltonians in magnets: Theories and computations". In: *Molecules* 26 (4 Feb. 2021).

[8] E. K. Delczeg-Czirjak et al. "Origin of the magnetostructural coupling in $FeMnP_{0.75}Si_{0.25}$". In: *Physical Review B* 90 (21 Dec. 2014), p. 214436.

[9] R Logemann et al. "Exchange interactions in transition metal oxides: the role of oxygen spin polarization". In: *Journal of Physics: Condensed Matter* 29 (33 Aug. 2017), p. 335801.

[10] I. V. Solovyev. "Exchange interactions and magnetic force theorem". In: *Physical Review B* 103 (10 Mar. 2021).

[11] Y. Mozharivskyj. *Magnetocaloric Effect and Magnetocaloric Materials*. Elsevier, 2016.

[12] K. A. Gschneidner and V. K. Pecharsky. "Magnetocaloric Materials". In: *Annual Review of Materials Science* 30 (1 Aug. 2000), pp. 387–429.

[13] Julia Lyubina. "Magnetocaloric materials for energy efficient cooling". In: *Journal of Physics D: Applied Physics* 50 (5 Jan. 2017).

[14] Hicham Johra et al. "Integration of a magnetocaloric heat pump in an energy flexible residential building". In: *Renewable Energy* 136 (June 2019), pp. 115–126.

[15] Xindong Wang et al. "Validity of the force theorem for magnetocrystalline anisotropy". In: *Journal of Magnetism and Magnetic Materials* 159 (3 July 1996), pp. 337–341.

[16] Christopher E. Patrick, Munehisa Matsumoto, and Julie B. Staunton. "First-principles calculations of the magnetocrystalline anisotropy of the prototype 2:17 cell boundary phase $Y(Co_{1-x-y}Fe_xCu_y)_5$". In: *Journal of Magnetism and Magnetic Materials* 477 (May 2019), pp. 147–155.

[17] Haruki Okumura et al. "First-principles Calculation of Magnetocrystalline Anisotropy of $Y(Co,Fe,Ni,Cu)_5$ Based on Full-potential KKR Green's Function Method". In: (Apr. 2022).

[18] Nicola Marzari et al. "Maximally localized Wannier functions: Theory and applications". In: *Reviews of Modern Physics* 84 (4 Oct. 2012), pp. 1419–1475.

[19] Richard Haberman. *Applied partial differential equations with fourier series and boundary value problems*. Pearson, 2014. ISBN: 9781292039855.

[20] F. S. Ham and B. Segall. "Energy Bands in Periodic Lattices—Green's Function Method". In: *Physical Review* 124 (6 Dec. 1961), pp. 1786–1796.

[21] Larry C. Grove. *Classical Groups and Geometric Algebra*. Ed. by Steven G. Krantz et al. The American mathematical society, 2002.

[22] Marcia Fampa and Jon Lee. "Efficient treatment of bilinear forms in global optimization". In: (Mar. 2018). URL: http://arxiv.org/abs/1803.07625.

[23] Jessica Gronski. "Non-Convex Optimization and Applications to Bilinear Programming and Super-Resolution Imaging". In: *ProQuest* 80 (9 2019).

[24] V. P. Antropov et al. "On the calculation of exchange interactions in metals". In: *Journal of Applied Physics* 99 (8 Apr. 2006), 08F507.

[25] M. Van Schilfgaarde and V. P. Antropov. "First-principles exchange interactions in Fe, Ni, and Co". In: *Journal of Applied Physics* 85 (8 II A Apr. 1999), pp. 4827–4829.

[26] T. Oguchi, K. Terakura, and A. R. Williams. "Band theory of the magnetic interaction in MnO, MnS, and NiO". In: *Physical Review B* 28 (11 Dec. 1983), pp. 6443–6452.

[27] Juan Rodríguez-Carvajal and Jacques Villain. "Magnetic structures". In: *Comptes Rendus Physique* 20 (7-8 Nov. 2019), pp. 770–802.

[28] Pierre Villars and Karin Cenzual, eds. *MnO Crystal Structure: Datasheet from "PAULING FILE Multinaries Edition – 2012" in SpringerMaterials*. URL: https://materials.springer.com/isp/crystallographic/docs/sd_1922219.

[29] X.B. Liu et al. "Fe magnetic moment formation and exchange interaction in Fe2P: A first-principles study". In: *Physics Letters A* 377 (9 Mar. 2013), pp. 731–735.

[30] Kristin Persson. *Materials Data on Fe2P (SG:189) by Materials Project*. July 2016.

[31] Qi Shen et al. "The antiferromagnetic to ferrimagnetic phase transition in Mn2Sb1-Bi compounds". In: *Journal of Alloys and Compounds* 866 (June 2021), p. 158963.

[32] Kristin Persson. *Materials Data on Mn2Sb (SG:129) by Materials Project*. Feb. 2016.

[33] Evgenii D. Chernov and Alexey V. Lukoyanov. "Magnetic Ground State and Electronic Structure of Binary Mn2Sb Compound from Ab Initio Calculations". In: MDPI AG, Dec. 2020, p. 15.

[34] R. Masrour et al. "Electronic and magnetic structures of ferrimagnetic Mn2Sb compound". In: *Journal of Magnetism and Magnetic Materials* 374 (Jan. 2015), pp. 116–119.

[35] Giovanni Pizzi et al. "Wannier90 as a community code: New features and applications". In: *Journal of Physics Condensed Matter* 32 (16 Apr. 2020).

[36] Xu He et al. "TB2J: a python package for computing magnetic interaction parameters". In: *Computer Physics Communications* (Sept. 2020). URL: http://arxiv.org/abs/2009.01910.

[37] G. Kresse and J. Furthmüller. "Efficient iterative schemes for *ab initio* total-energy calculations using a plane-wave basis set". In: *Physical Review B* 54 (16 Oct. 1996), pp. 11169–11186.

[38] G. Kresse and J. Furthmüller. "Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set". In: *Computational Materials Science* 6 (1 July 1996), pp. 15–50.

[39] G. Kresse and J. Hafner. "*Ab initio* molecular-dynamics simulation of the liquid-metal–amorphous-semiconductor transition in germanium". In: *Physical Review B* 49 (20 May 1994), pp. 14251–14269.

[40] G. Kresse and J. Hafner. "*Ab initio* molecular dynamics for liquid metals". In: *Physical Review B* 47 (1 Jan. 1993), pp. 558–561.

[41] Andreas Wächter and Lorenz T. Biegler. “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming”. In: *Mathematical Programming* 106 (1 Mar. 2006), pp. 25–57.

[42] Maame Yaa B. Poku, Lorenz T. Biegler, and Jeffrey D. Kelly. “Nonlinear Optimization with Many Degrees of Freedom in Process Engineering”. In: *Industrial & Engineering Chemistry Research* 43 (21 Oct. 2004), pp. 6803–6812.

[43] Logan Beal et al. “GEKKO Optimization Suite”. In: *Processes* 6 (8 July 2018), p. 106.

[44] John D. Hedengren et al. “Nonlinear modeling, estimation and predictive control in APMonitor”. In: *Computers and Chemical Engineering* 70 (Nov. 2014), pp. 133–148.

Appendix A: 3D representation of boundary conditions

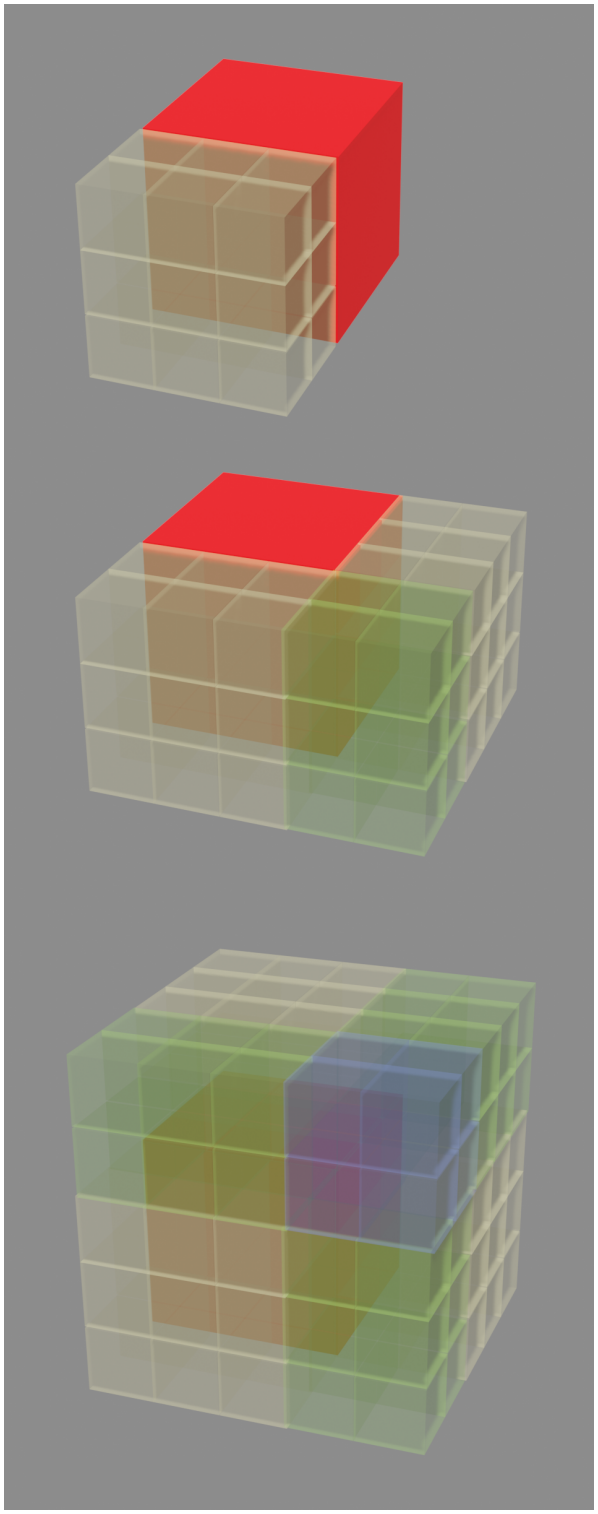


Figure 17: 3D representation of the boundary conditions. The original 3×3×3 supercell is shown in red. Added areas of types $C_j(l)$, $D_j(m)$ and E_j are shown in translucent yellow, green and blue, respectively. From top to bottom the images represent the added exchange interactions for a face, edge and corner, respectively.

Appendix B: Symmetry finding algorithm flowchart

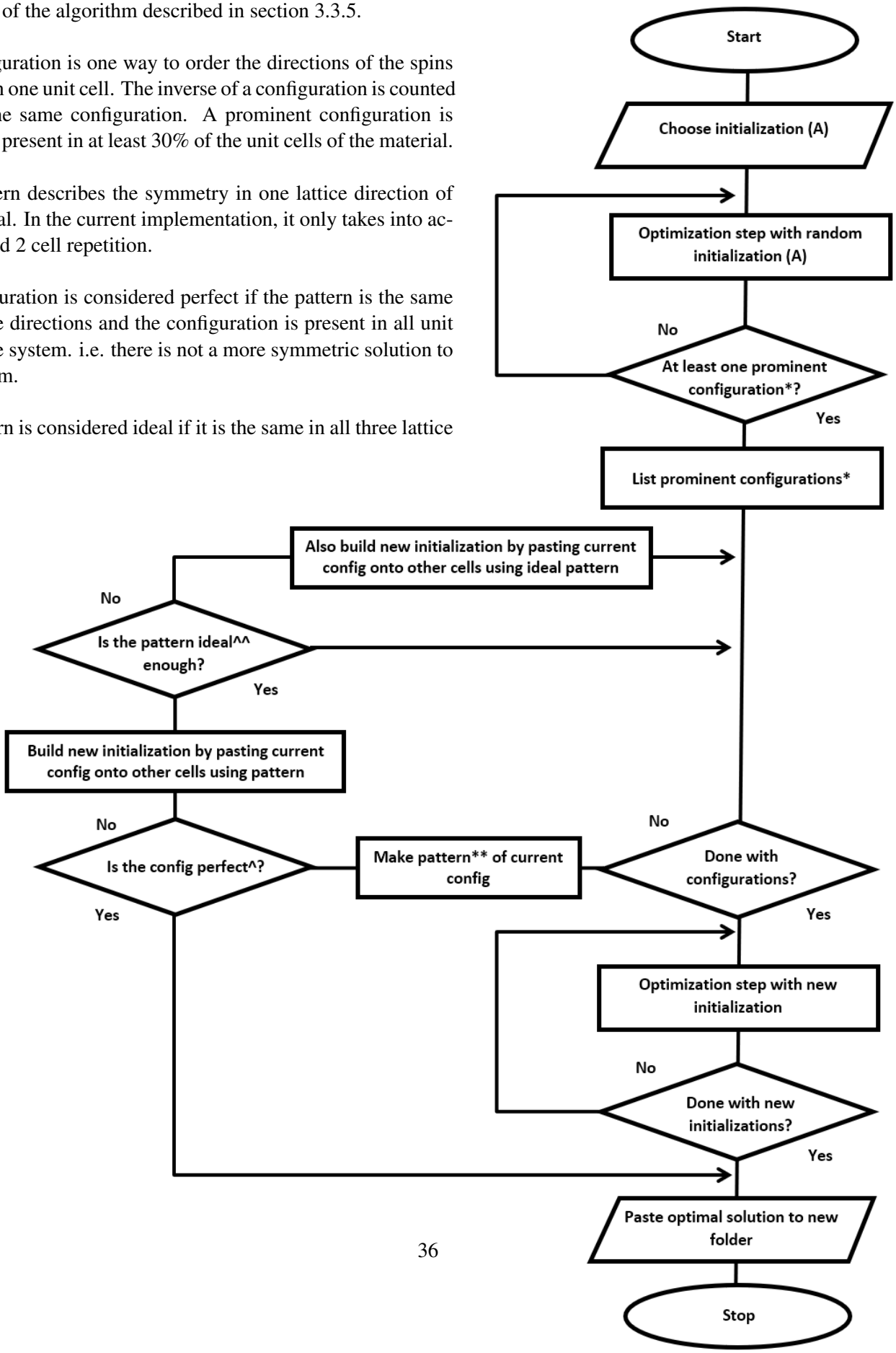
Flowchart of the algorithm described in section 3.3.5.

* A configuration is one way to order the directions of the spins of atoms in one unit cell. The inverse of a configuration is counted towards the same configuration. A prominent configuration is one that is present in at least 30% of the unit cells of the material.

** A pattern describes the symmetry in one lattice direction of the material. In the current implementation, it only takes into account 1 and 2 cell repetition.

ˆ A configuration is considered perfect if the pattern is the same in all three directions and the configuration is present in all unit cells of the system. i.e. there is not a more symmetric solution to the problem.

ˆ ˆ A pattern is considered ideal if it is the same in all three lattice directions.



Appendix C: Auxiliary results

In this appendix, the convergence plots, and some suboptimal configurations of the different executed runs are presented. There are 11 main figures, all spanning a page. First, the full convergence plots of the runs executed for the basic model, and the basic model including the boundary conditions are presented (figures 18 and 19). Next, the iterations of the different runs of the algorithm on the test set are shown in figures per material, as to compare the different values of γ (figures 20 - 26). Lastly, some frequently occurring suboptimal configurations for MnO are shown in figures 27 and 28.
For the convergence plots, note that some iterations and final exchange energies may overlap, hence the total amount of runs that end at the same energy level is not entirely evident from these plots alone. For an overview of the amount of successful runs, we refer back to the tables shown in section 4.
For the configurations plots, these are by no means all configurations found in the runs. They are presented here only to serve as examples.

Energy differences

Due to the construction of the model, the two figures 13 and 14 contain quite different total energies. The first difference is that between the $2\times2\times2$ and $3\times3\times3$ configurations for the basic and boundary conditions model among itself. This effect can be explained by the order of exchange interactions that is taken into account. For a $3\times3\times3$ cell, more exchange interactions are evaluated than in a $2\times2\times2$ cell. Hence, a noticeable difference in total energy appears between the two configurations. The energy difference between the model with and without boundary conditions, is due to how the boundary conditions are implemented. Because the boundary conditions add equations to the total energy function, there are more exchange interactions taken into account (those that for the basic model would fall outside of the supercell). This results in a significantly higher energy for both configurations. The $2\times2\times2$ case **benefits more** from this effect than the $3\times3\times3$ case, as when we look back at equations (31, 32 and 33), way more boundary conditions are added for a corner than for a side or face unit cell. As the $2\times2\times2$ only consists of corners, and the $3\times3\times3$ does not (its outer shell consists of 8 corners, 12 edges and 6 faces), it so to speak 'gains' less net equations. Hence, for the model with boundary conditions, the $3\times3\times3$ configurations have a consistently higher energy per unit cell, where for the base model the differences are smaller, and the $3\times3\times3$ case has a lower energy than the $2\times2\times2$ case.

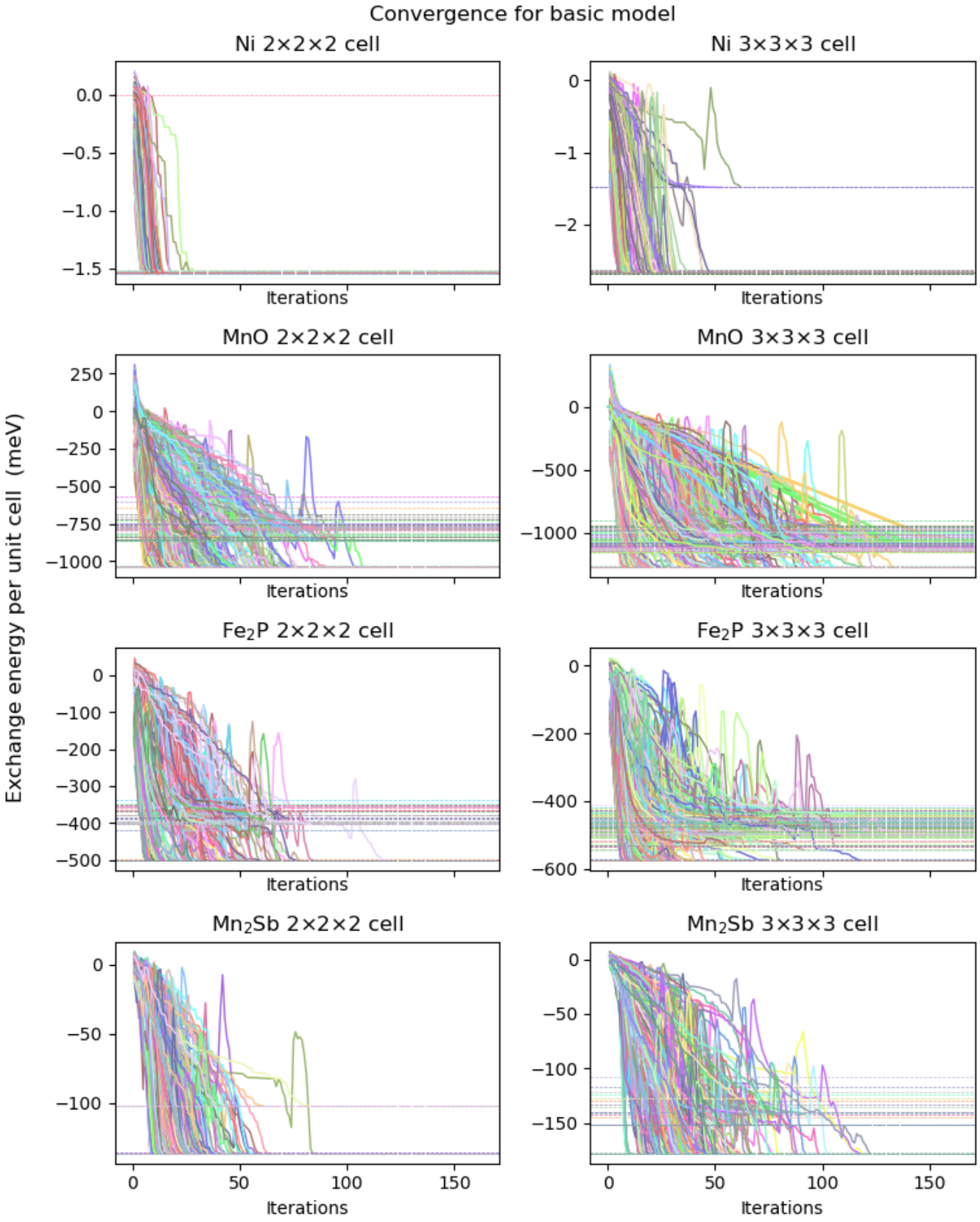


Figure 18: The full iterations for all runs executed in testing the basic model. Different runs are shown in different colours. In this plot, no distinction is made between different initializations.

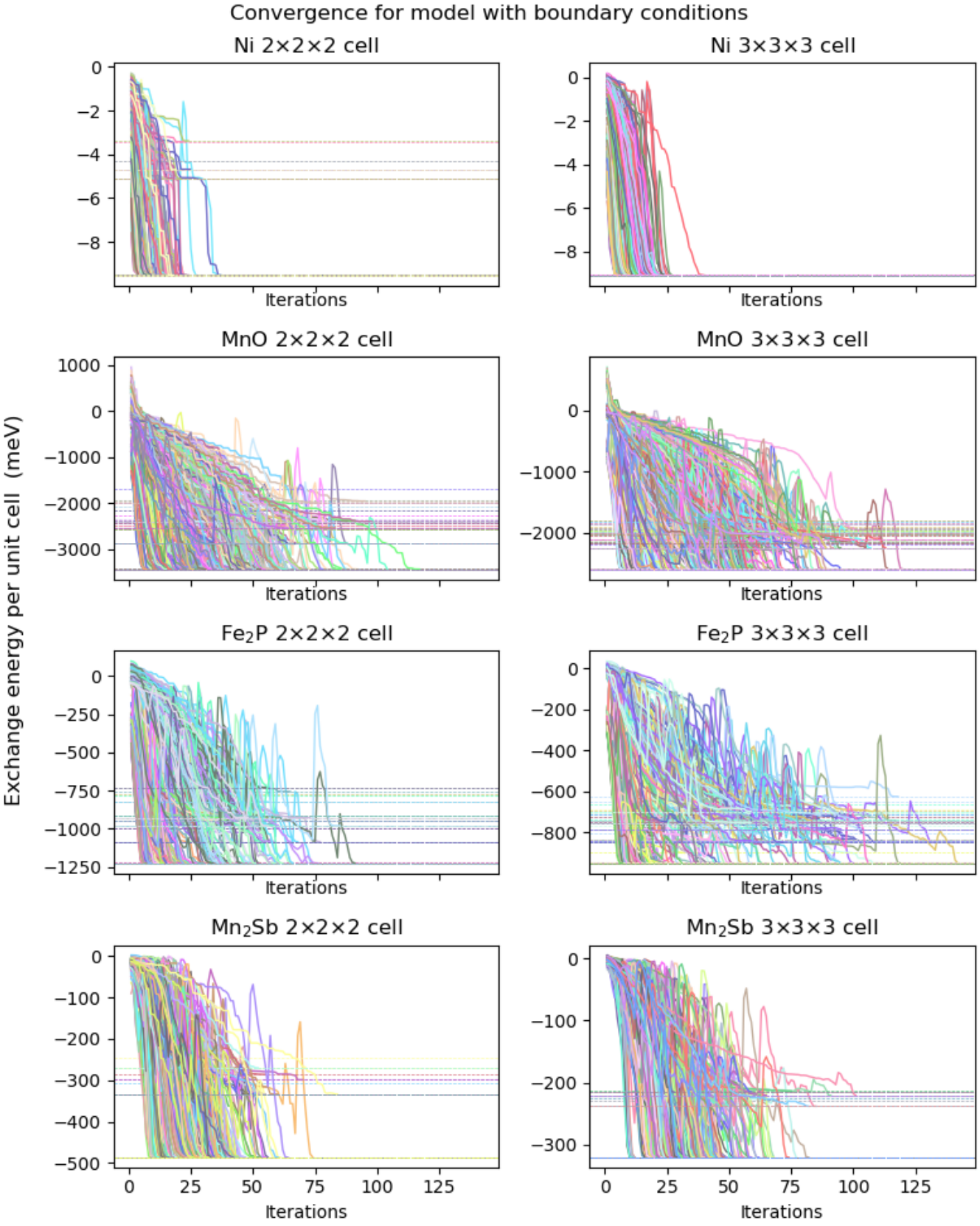


Figure 19: The full iterations for all runs executed in testing the basic model with boundary conditions. Different runs are shown in different colours. In this plot, no distinction is made between different initializations.

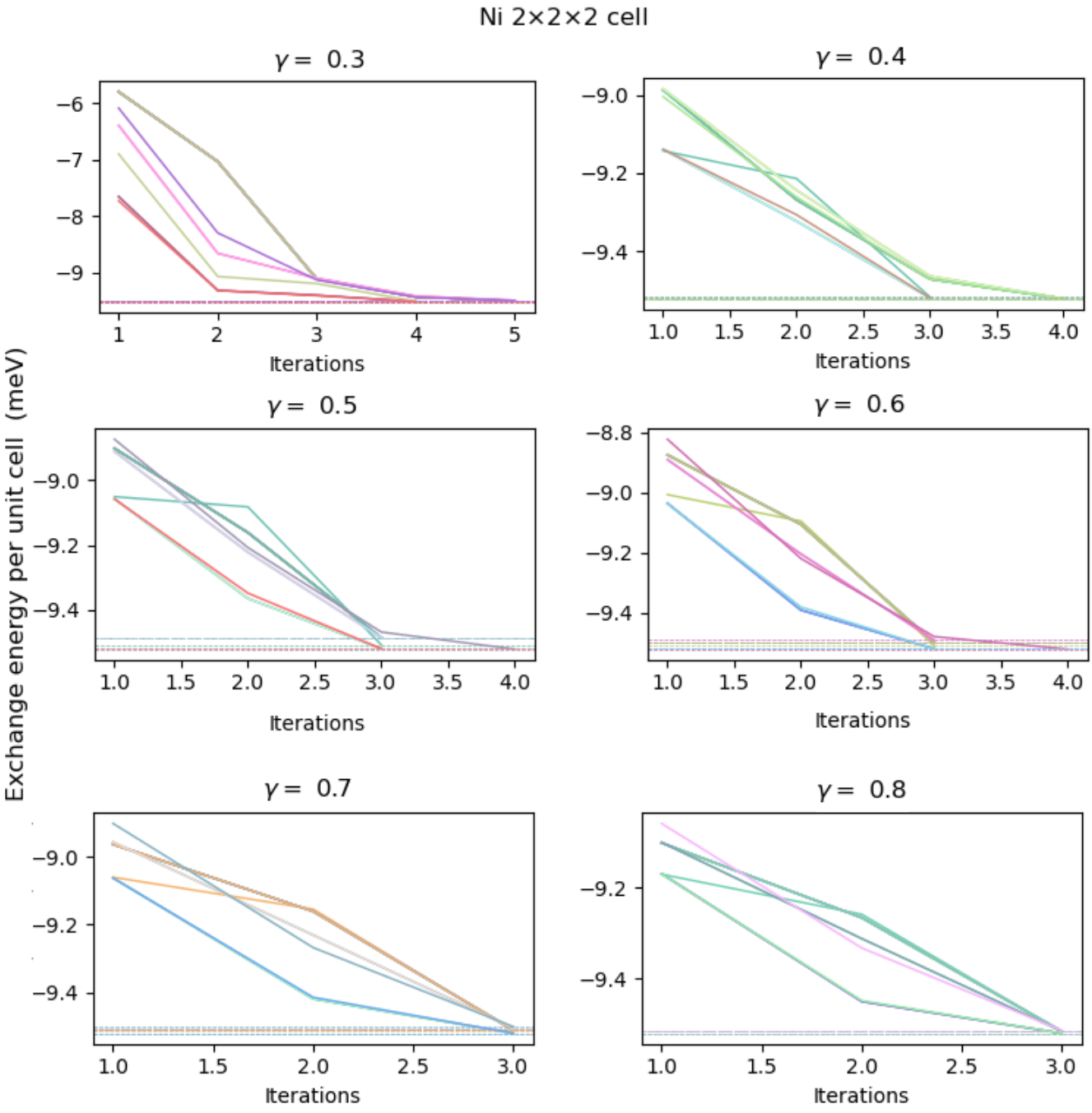


Figure 20: The full iterations for all runs executed on the test set of Ni 2×2×2.

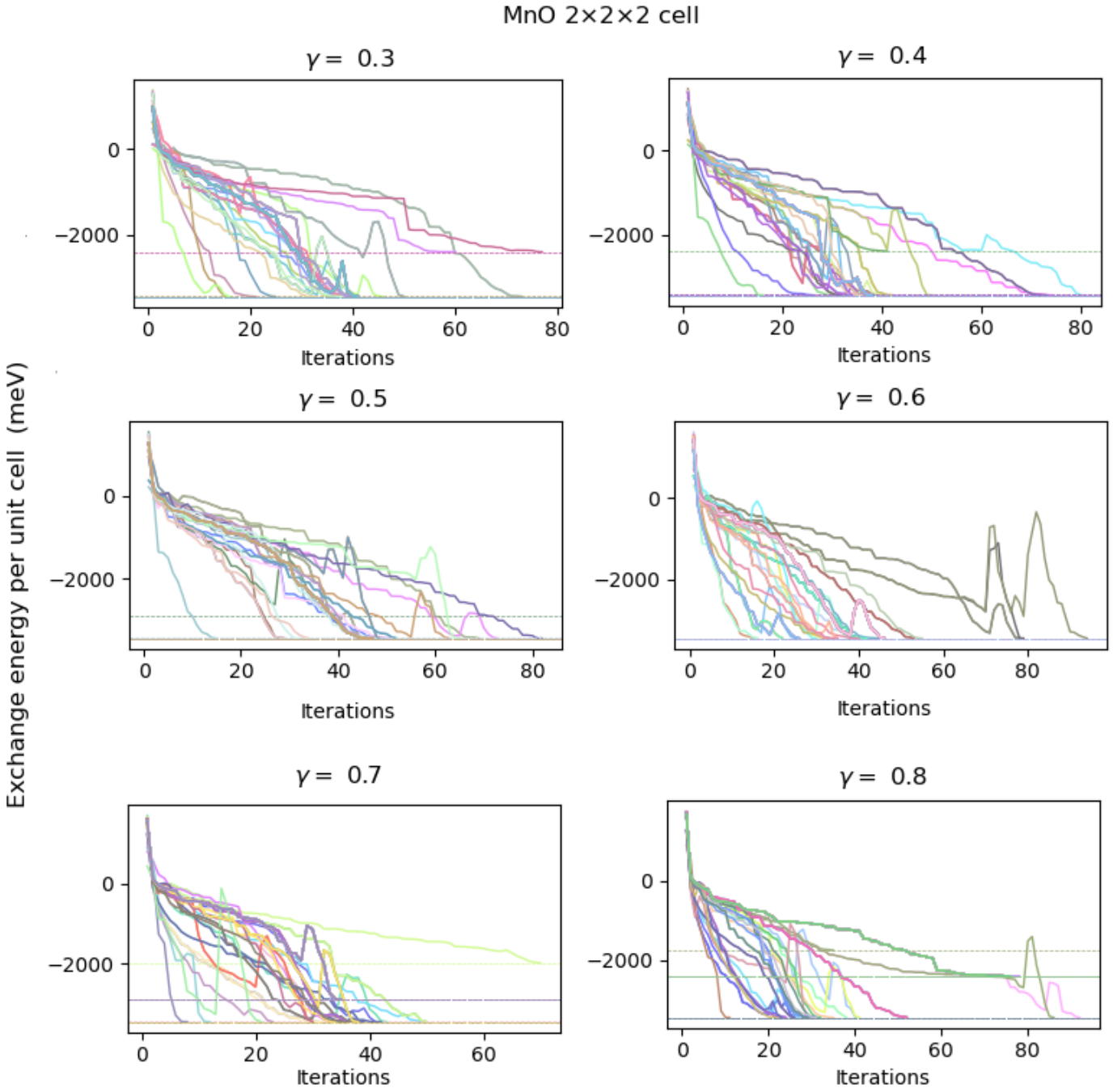


Figure 21: The full iterations for all runs executed on the test set of MnO 2×2×2.

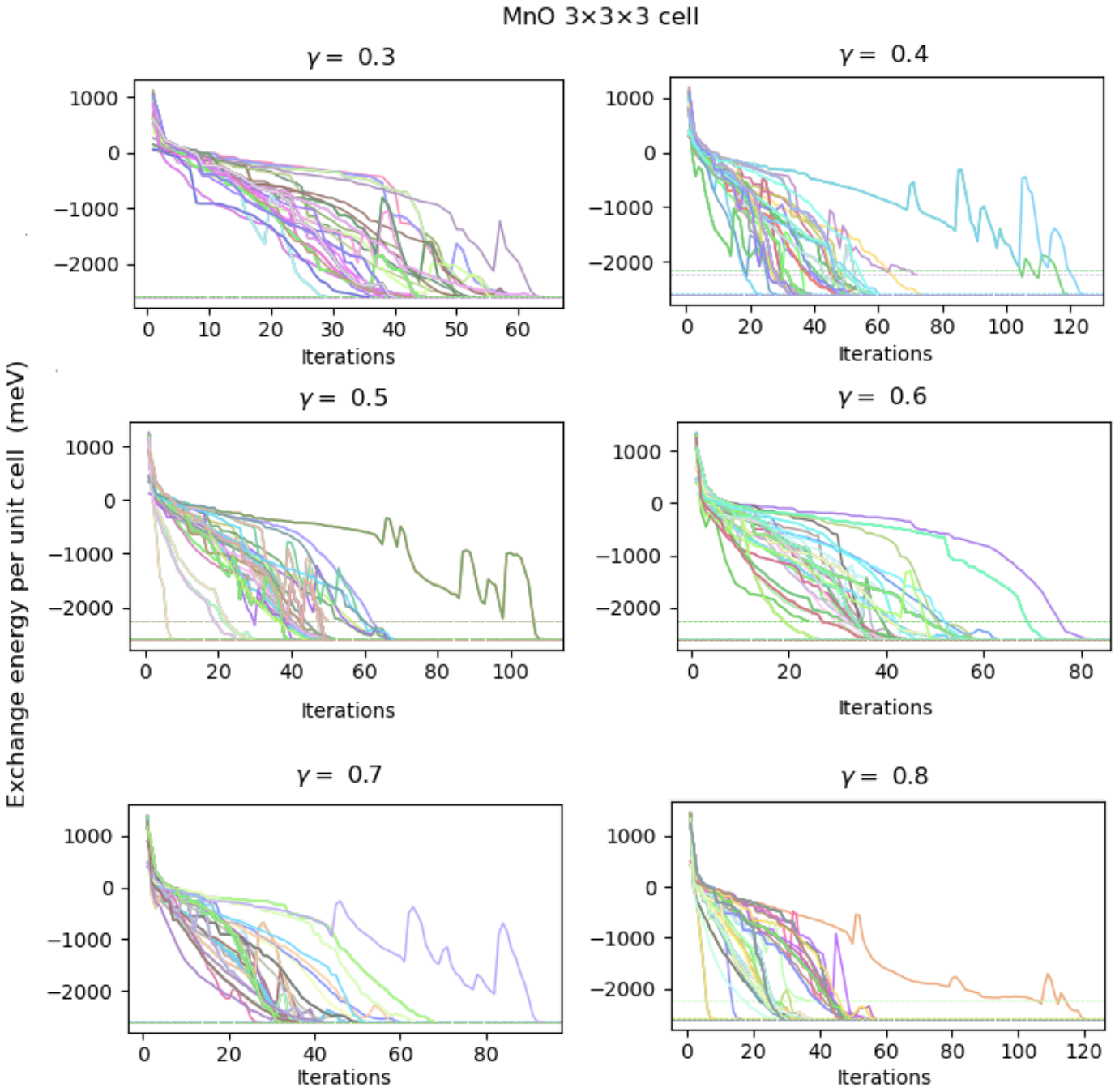


Figure 22: The full iterations for all runs executed on the test set of MnO 3×3×3.

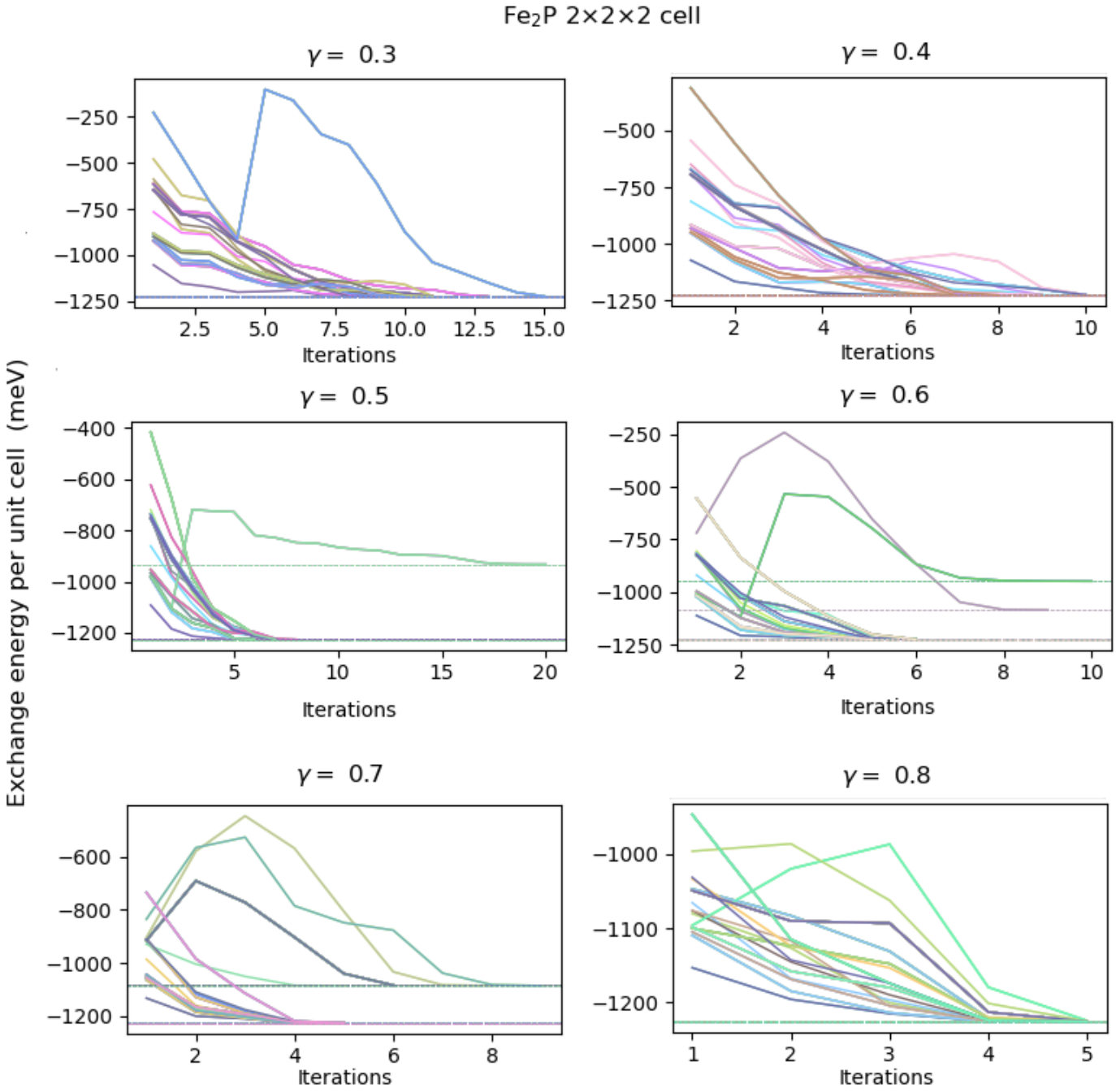


Figure 23: The full iterations for all runs executed on the test set of Fe_2P 2×2×2.

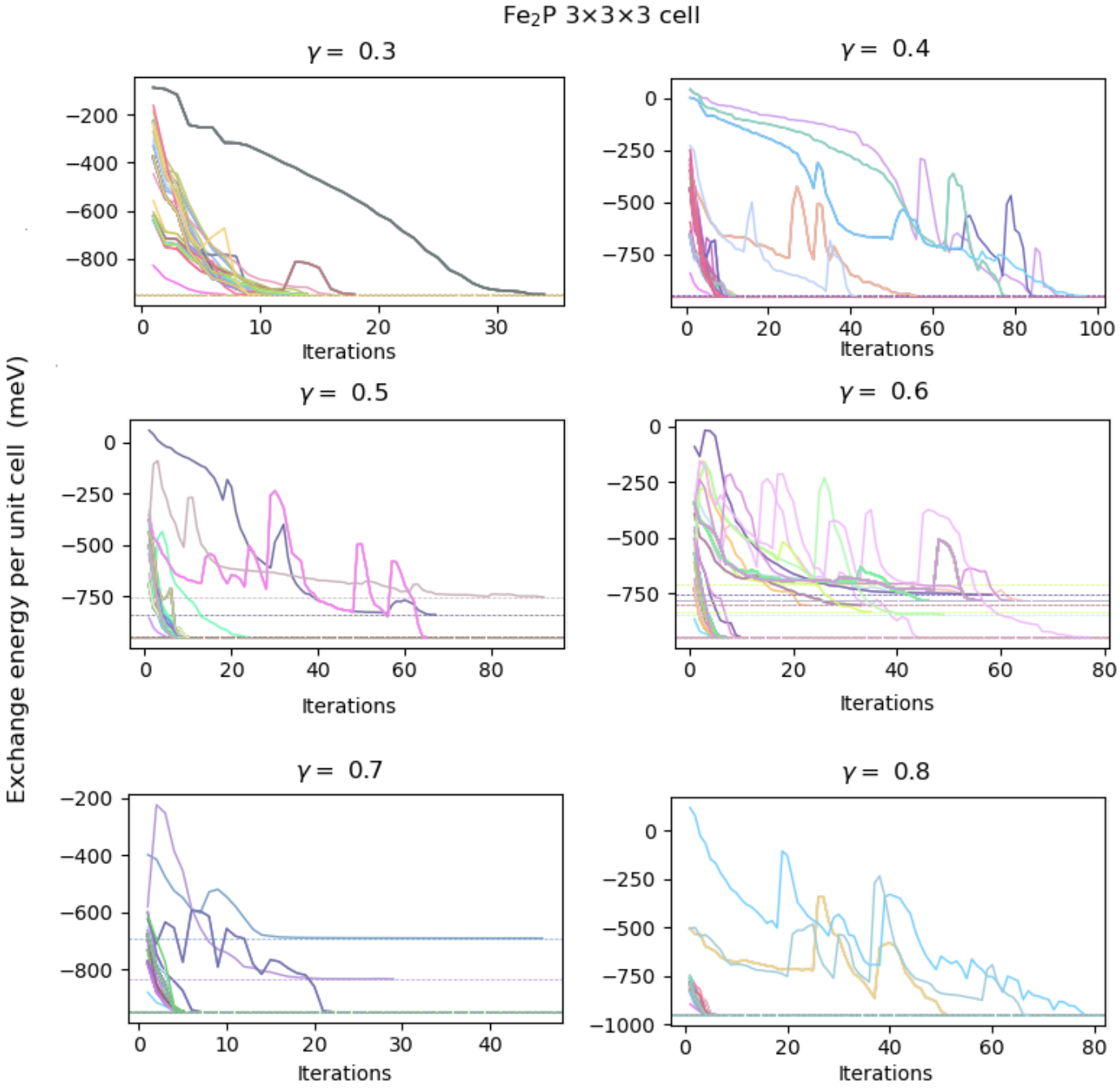


Figure 24: The full iterations for all runs executed on the test set of Fe_3P 3×3×3.

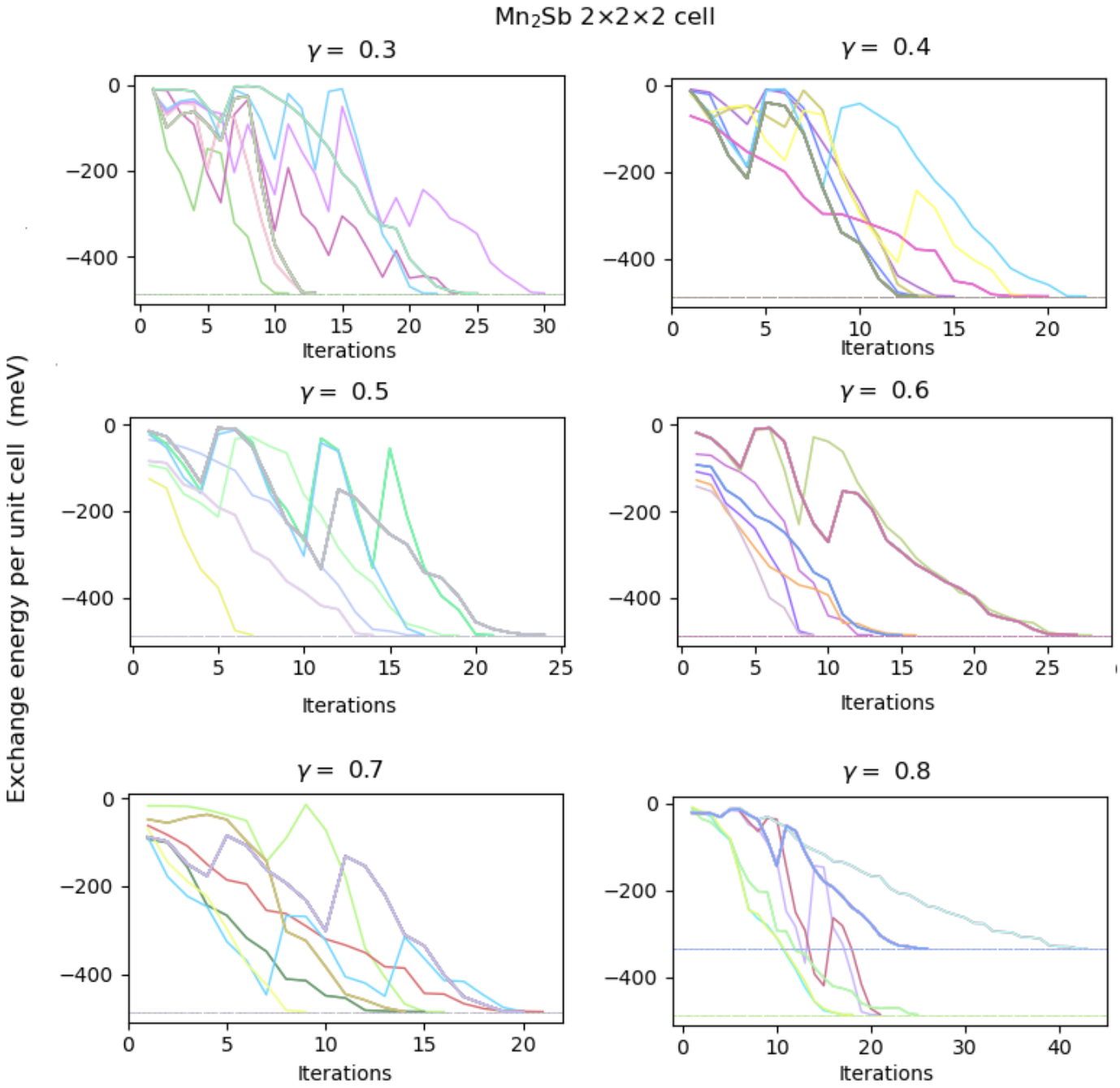


Figure 25: The full iterations for all runs executed on the test set of Mn_2Sb 2×2×2.

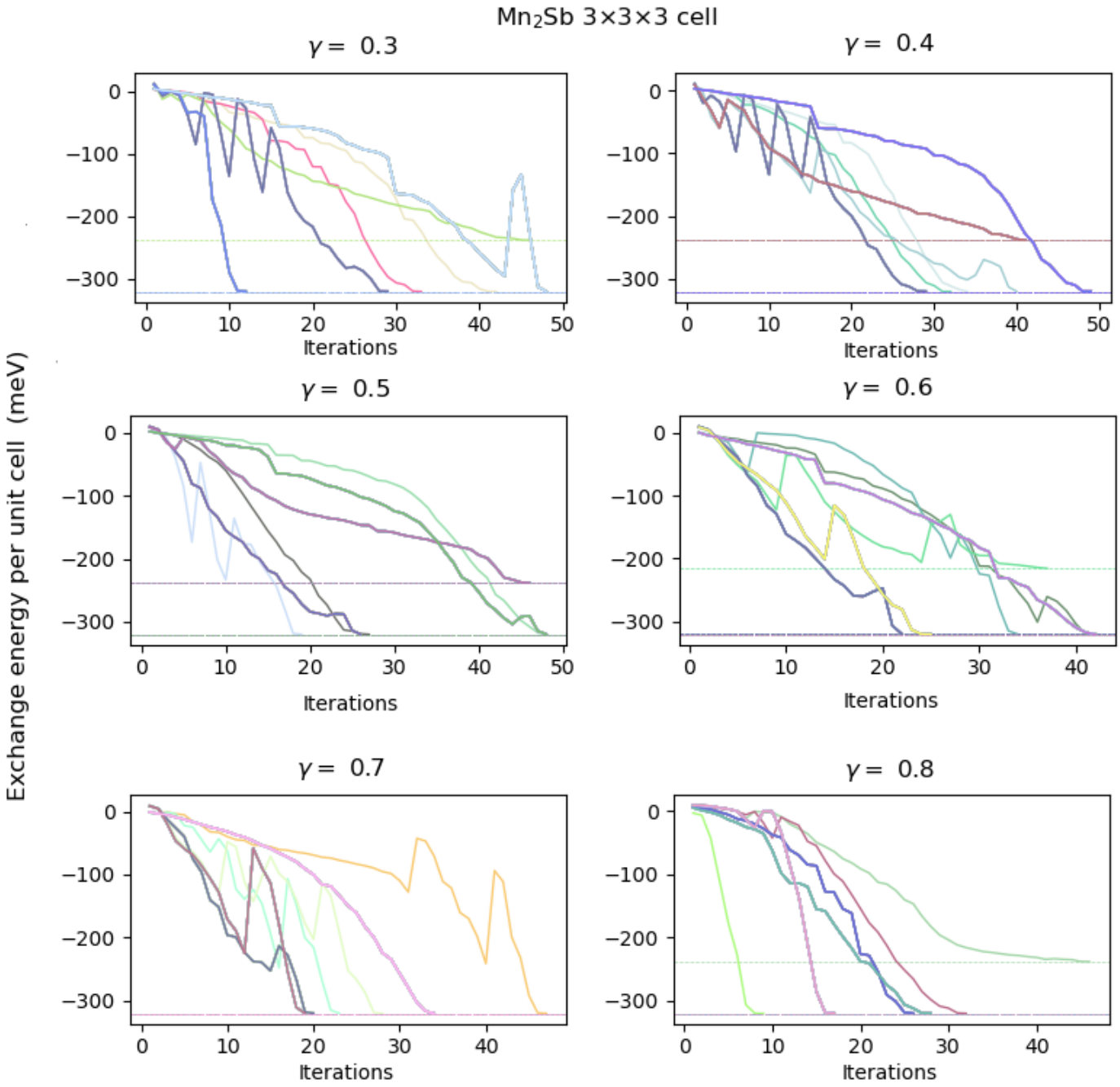


Figure 26: The full iterations for all runs executed on the test set of Mn_2Sb 3×3×3.

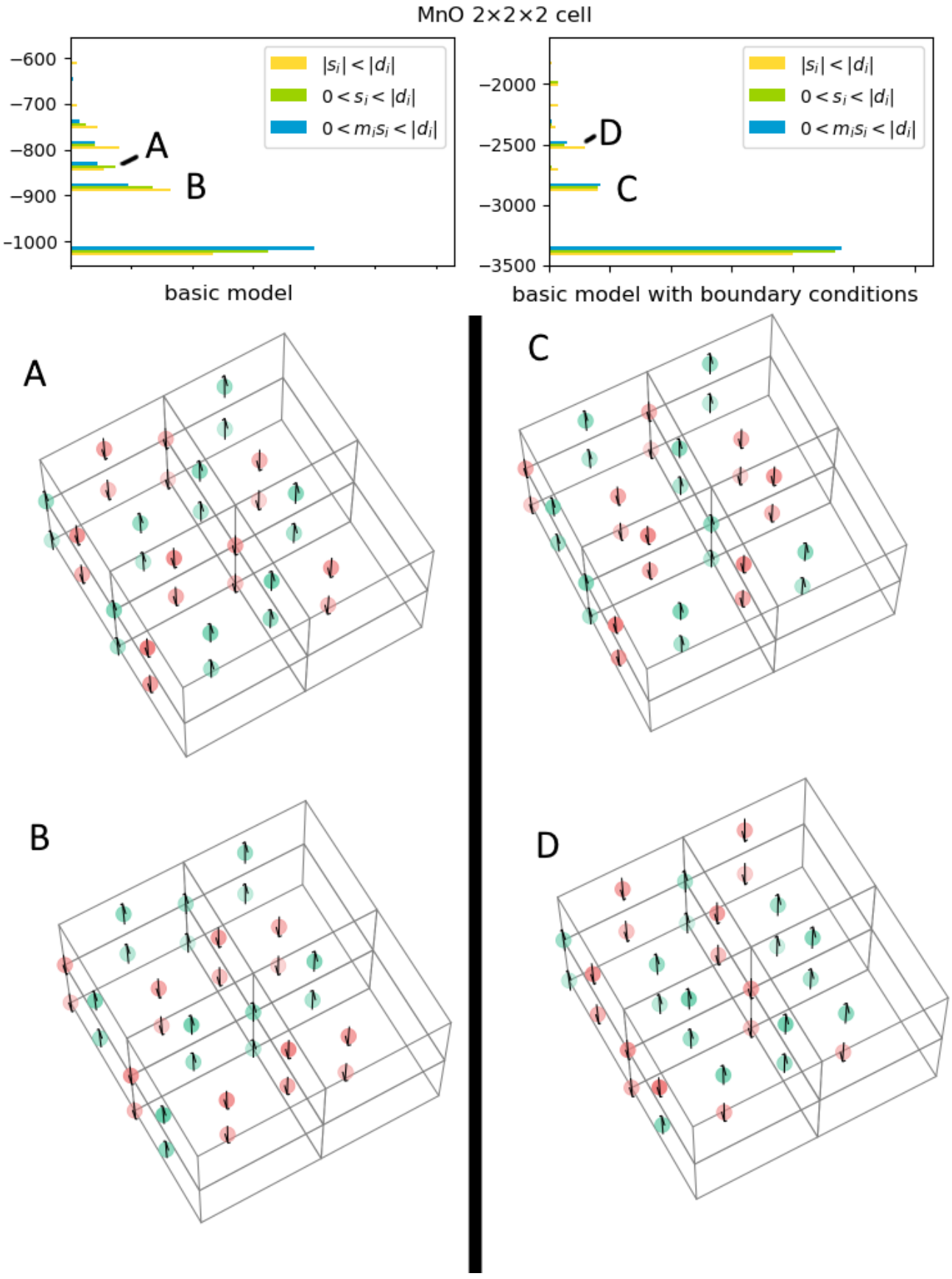


Figure 27: Two suboptimal configurations for the basic model, and the model with boundary conditions, for MnO 2×2×2. The different energies corresponding to the different configurations are marked by A, B, C and D.

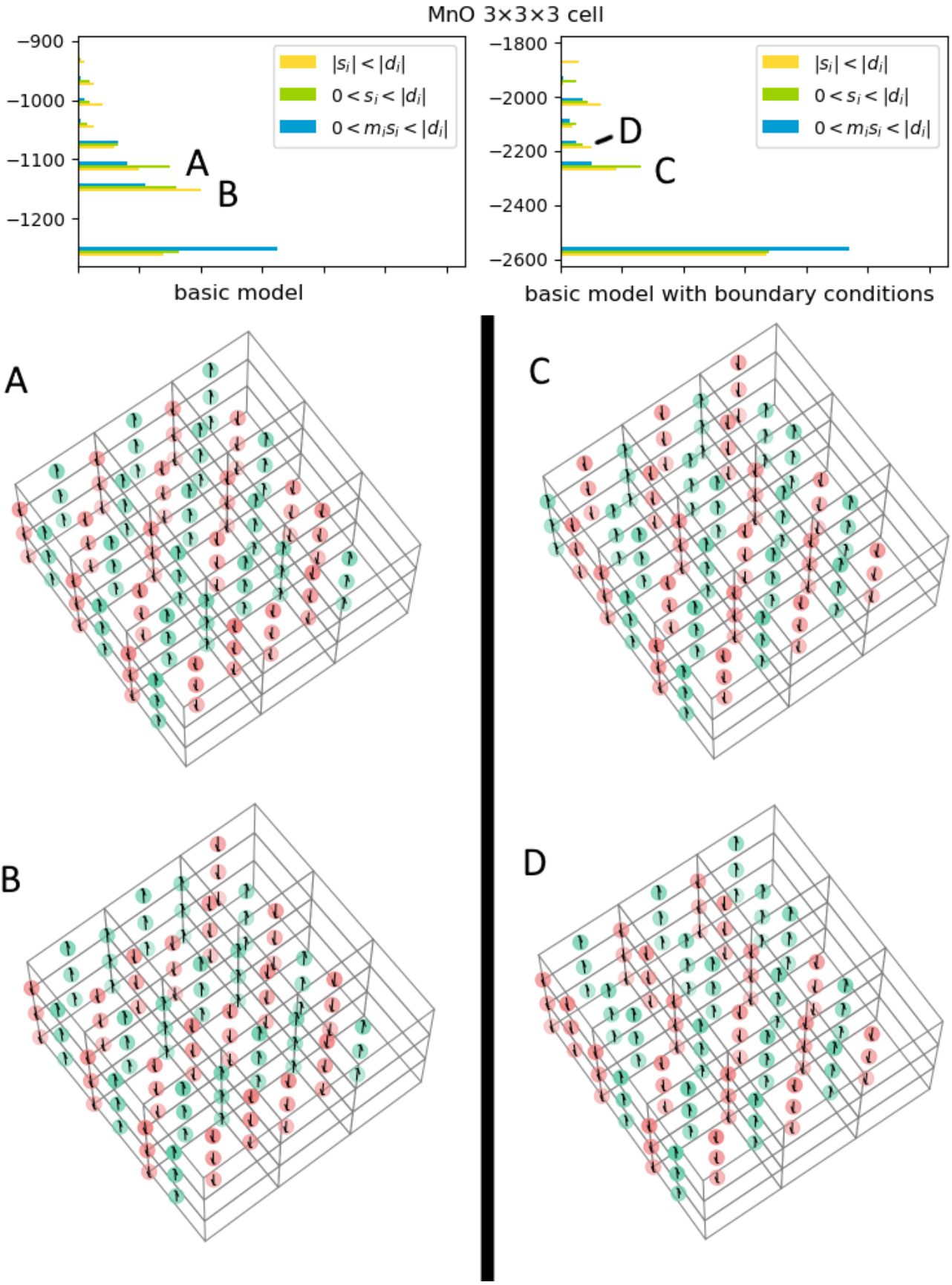


Figure 28: Two suboptimal configurations for the basic model, and the model with boundary conditions, for MnO 3×3×3. The different energies corresponding to the different configurations are marked by A, B, C and D.

Appendix D: Python code

```
# -*- coding: utf-8 -*-
"""
Created on Thu May 19 21:09:22 2022

@author: Arthur C. Ronner
"""
# -*- coding: utf-8 -*-

from pymatgen.io.vasp.inputs import Poscar
from pymatgen.core import lattice
import numpy as np
import re
import os
from pathlib import Path
from gekko import GEKKO
from contextlib import redirect_stdout
import mag_plotter as mp
import time
import shutil

#==================================#
#                                  #
# FILE INDEX:                      #
#   files & formating              #
#       READ FILE FUNCTION         #
#       FORMATTING FUNCTIONS       #
#                                  #
#   One opt run                    #
#       PERIODIC BC'S              #
#       VARIABLE INITIALIZATION    #
#       DEFINING OPTIMIZATION      #
#       SMALL HELP FUNCTIONS       #
#       MAIN RUNNER FUNCTIONS      #
#                                  #
#   Multiple runs                  #
#       MULTIPLE RUNS CODE         #
#       ITERATING FUNCTIONS        #
#                                  #
#==================================#

#=====================================================================#
#                         READ FILE FUNCTIONS                         #
#=====================================================================#
def read_file(file_name,num_hor_vals, file_type, convert_names = {}):
    """
    :file_type: -1 = J from jdata file; 0 = J matrix short; 1 = results from Gekko;
↪   2 = APOPT; 3 = IPOPT; 4 = mag_moment file
    """
    #print(file_name)
    with open(file_name) as f:
        lines = f.readlines()
    #Predict filesize horizontal
    data = np.zeros((len(lines),num_hor_vals)) #initialize data vector
    if file_type == 0 or file_type == -1:   #As each file is a little diffferent we
    ↪  prepare different arrays
        names = [['', ''] for i in range(len(lines))] #here we need 2 names as it
        ↪  is J interactions
    if file_type == 1:
```

```
        names2 = ['' for i in range(len(lines))] #here only one as its about
        ↪  variables

    for i in range(len(lines)): #per file we format lines a little differently so
    ↪  that we get the correct data
        if file_type == 0:
            lines[i] = re.sub(',|\(|\)', '', lines[i])#remove comma's
        if file_type == 1:
            lines[i] = re.sub('\{|\}', "0 0 0 0", lines[i])
            lines[i] = re.sub(',|\[|\]|:|\"', '', lines[i])
            lines[i] = re.sub('q_', ' -', lines[i])
            lines[i] = re.sub('q', ' ', lines[i])
            lines[i] = lines[i].title()
        if file_type == 4:
            lines[i] = re.sub('\s[a-z]', '_',lines[i])
        data_frag = lines[i].split()
        #print(lines[i])

        for j in range(num_hor_vals):

            try:
                float(data_frag[j])
                data[i,j] = float(data_frag[j])
                if file_type == -1:
                    if j <= 1:
                        names[i][j] = convert_names[int(data[i,j]-1)]
            except ValueError:
                if file_type ==0:
                    if j == 0:
                        names[i][j] = re.sub(r'(-?[0-9]+\.?[0-9]*)' , r" \1
                        ↪  ",data_frag[j].strip())
                        names[i][j] = re.sub('\s$','',names[i][j])
                    elif j == 1:
                        names[i][j] = re.sub(r'(-?[0-9]+\.?[0-9]*)' , r" \1
                        ↪  ",data_frag[j].strip()) #capitalizes letters
                        names[i][j] = re.sub('\s$','',names[i][j])
                if file_type == 1:
                    if j == 0:
                        names2[i] = re.sub('_', ' ',data_frag[j])
                continue
            except IndexError:
                continue
    if file_type == 0:
        return data[:,2:], np.array(names)
    elif file_type == 1:
        l = 2
        for i in range(l,len(names2)):
            if len(re.findall('^P\d',names2[i])) == 0:
                l = i
                break
        return data[l:-1,1:], names2[l:-1]
    elif file_type == 2:
        return data[1:-2,:]
    elif file_type == 3:
        return data[1:-3,:]
    elif file_type == -1:
        return data[:,[2,3,4,5]], names
    else:
        return data

```

```
112 def read_moments(file_name, TB2J = True, N = 100):
113
114     if TB2J:
115         data = [[] for i in range(N)]
116         with open(file_name, "r") as file:  # the r opens it in read mode
117             count = 0
118             for i in file:
119                 count += 1
120             N = min(count-3,N)
121         with open(file_name, "r") as file:
122             print(N)
123             for i in range(N):
124                 line = next(file).strip()
125                 if len(re.findall('Atoms', line)) > 0:
126                     start = i +3
127                 if len(re.findall('^Total',line)) > 0:
128                     stop = i
129                 data[i] = line
130         interest = [[] for i in range(stop-start)]
131         for i in range(start,stop):
132             interest[i-start] = data[i].split()
133         moments = {}
134         for i in range(stop-start):
135             moments[re.sub(r'(-?[0-9]+\.?[0-9]*)' , r" \1",interest[i][0].strip())]
                ↪  = float(interest[i][-1])
136         return moments
137     else:
138         moments = {}
139         names = []
140         with open(file_name, "r") as f:
141             lines = f.readlines()
142             for i in range(len(lines)):
143                 temp = lines[i].split()
144                 if len(temp) != 2:
145                     print('ERROR!')
146                 moments[re.sub(r'(-?[0-9]+\.?[0-9]*)' , r" \1",temp[0].strip())] =
                    ↪  float(temp[1])
147                 names.append(re.sub(r'(-?[0-9]+\.?[0-9]*)' , r"
                    ↪  \1",temp[0].strip()))
148         return moments, names
149 #=====================================================================#
150 #                        FORMATTING FUNCTIONS                         #
151 #=====================================================================#
152
153 def J_short(file, new_file, ftype = 0):
154     with open(file) as f:
155         lines = f.readlines()
156     g = open(new_file, "w")
157     start = True
158     stop = False
159     if ftype == 0:
160         for i in lines:
161             x = re.findall("^\-\-\-|J_iso|Orbital| \[", i)
162             if x == ['---']:
163                 start = False
164             if not x and start == False:
165                 g.write(i)
166     elif ftype == 1: #APOPT
167         for i in lines:
168             x = re.findall('Iter|Success',i)
```

```
169
170             if x == ['Iter']:
171                 start = False
172             if x == ['Success']:
173                 stop = True
174             if not x and start == False and stop == False and len(i)>10:
175                 g.write(i)
176     elif ftype == 3: #IPOPT
177         for i in lines:
178             x = re.findall('^iter|^Number of I',i)
179             if x == ['iter']:
180                 start = False
181             if x == ['Number of I']:
182                 stop = True
183             if not x and start == False and stop == False:
184                 g.write(i)
185     g.close()
186     return
187
188
189
190 def J_as_dict(J_mat,nam_mat,latt, index):
191     J_dict_lvec = {}
192     name_dict = {}
193     #names = np.unique(nam_mat)
194     old_s = str(latt.sites[0].specie)
195     j = 1
196     for i in range(len(latt.sites)):
197         new_s = str(latt.sites[i].specie)
198         if new_s != old_s:
199             j = 1
200         name_dict[str(latt.sites[0].specie) + ' '+ str(j)] =
        ↪  latt.sites[i].coords.dot(latt.lattice.inv_matrix)
201         j += 1
202         old_s = str(latt.sites[i].specie)
203     for i in range(len(J_mat)):
204         nam = ['','']
205         for j in range(2):
206             nam[j] = nam_mat[i][j]
207             nam[j] = re.sub('\s$','',nam[j])
208         J_dict_lvec[nam[0]+' '+nam[1]+' '+str(int(J_mat[i,0]))+'
        ↪  '+str(int(J_mat[i,1]))+' '+str(int(J_mat[i,2]))] = J_mat[i,index]
209     #print('Done with saving Js as a dictionary')
210     return J_dict_lvec, name_dict
211
212
213
214 #=====================================================================#
215 #                        PERIODIC BC'S                                #
216 #=====================================================================#
217 def periodic_bc_2(m,J_mat, uni_at, var_vec,ind,ranges, J_reach):
218     tot = len(var_vec)
219     edge_num = len(var_vec) - (len(uni_at)*(ranges[1]-ranges[0] -
        ↪  1)*(ranges[3]-ranges[2] - 1)*(ranges[5]-ranges[4] -1))
220     print('number of atoms at edge: ' + str(edge_num))
221     bc_fct = [None for j in range(tot)]
222     cell =
        ↪  np.array([ranges[1]-ranges[0]+1,ranges[3]-ranges[2]+1,ranges[5]-ranges[4]+1])
223     k = 1
224     for i in range(len(var_vec)):
```

```
        if var_vec[i] is None:
            continue
        s_p = np.full(3,None) #s_p = saved points
        for j in range(3):
            if ind[i][j] == ranges[2*j] or ind[i][j] == ranges[2*j+1]:
                temp = ind[i].copy()
                if temp[j] == ranges[2*j]:
                    temp[j] = ranges[2*j+1]
                else:
                    temp[j] = ranges[2*j]
                orient = np.ones(4)
                orient[3] = 0 #setting mode
                orient[j] = 0 #setting axis to slice away
                k = ind.index([temp[0],temp[1],temp[2],temp[3]])
                bc_fct[i] = cut_equation(m,J_mat,uni_at,var_vec,ind,J_reach,
                ↪  cell,0,tot,i,typ = 1, j = k, ori = orient)
                s_p[j] = temp[j]
        u_c = np.where(s_p == None)[0] #u_c = unaltered coords
        if len(u_c) == 1: #we're at an edge:
            s_p[int(u_c)] = ind[i][int(u_c)]
            orient = np.zeros(4)
            orient[3] = 1
            orient[int(u_c)] = 1
            k = ind.index([s_p[0],s_p[1],s_p[2],ind[i][3]])
            bc_fct[i] =
            ↪  cut_equation(m,J_mat,uni_at,var_vec,ind,J_reach,cell,0,tot,i,typ =
            ↪  1, j = k, ori = orient)
        elif len(u_c) == 0: #we're at a corner, so need 4 other configs
            for l in range(3): #first three
                temp = np.zeros(3)
                orient = np.ones(4)
                orient[l-1] = 0
                orient[l-2] = 0 # ori will have 0's at axis that need 'slicing' and
                ↪  1 at unaltered axes.
                temp[l-1] = s_p[l-1]
                temp[l-2] = s_p[l-2]
                temp[l] = ind[i][l]
                k = ind.index([temp[0],temp[1],temp[2],ind[i][3]])
                bc_fct[i] =
                ↪  cut_equation(m,J_mat,uni_at,var_vec,ind,J_reach,cell,0,tot,i,typ
                ↪  = 1, j = k, ori = orient)

            k = ind.index([s_p[0],s_p[1],s_p[2],ind[i][3]]) #fourth
            orient = [0,0,0,2]
            bc_fct[i] = cut_equation(m,J_mat,uni_at,var_vec,ind,J_reach,cell,
            ↪  0,tot,i,typ = 1, j = k, ori = orient)


    return [g for g in bc_fct if g is not None]

def J_sum_cond(coords,ori): #avoids double counting in periodic bc's
    #ori = [x,y,z,mode] if x,y,z = 0 then these axis are to be cut away, 1 means
    ↪  they need to be retained.
    if ori[3] == 0: #case of a face
        axis = np.where(ori == 0)[0]
        return coords[axis[0]] != 0 #we remove one plane
    if ori[3] == 1:  #case of an edge
        axis = np.where(ori == 0)[0]
        #print(axis)
        return (coords[axis[0]] != 0 and coords[axis[1]] != 0) #remove two planes
```

```
277     if ori[3] == 2: #case of a corner
278         return (coords[0] != 0 and coords[1] != 0 and coords[2] != 0) #remove 3
            ↪ planes
279     return
280
281
282
283 #=====================================================================#
284 #                         VARIABLE INITIALIZATION                     #
285 #=====================================================================#
286
287 def initial_vals(ranges,unique_ats,moments,itype, at_index = []):
288
289     a, b, c, d, r = len(unique_ats), len(ranges[0]), len(ranges[1]),
        ↪ len(ranges[2]), range(len(unique_ats))
290     ms = np.zeros(a)
291     for i in r:
292         ms[i] = moments.get(unique_ats[i]) #To shorten notations
293
294     if itype >= 0: #These are the random seeds
295         vals = np.random.rand(a,b,c,d)
296         if itype == 0: #Totally random within bounds (for antiferromagnetic?)
297             for i in r:
298                 vals[i] = vals[i]*(2*abs(ms[i]))-abs(ms[i])
299             print("Initializing any random value within bounds")
300         if itype == 1: #Random positive values
301             for i in r:
302                 vals[i] = vals[i]*(abs(ms[i]))
303             print("Initializing any positive random value within bounds")
304         if itype == 2: #Random pos or neg depending on moment
305             for i in r:
306                 vals[i] = vals[i]*(ms[i])
307             print("Initializing value between 0 and the value in exchange.out (pos
                ↪ or neg)")
308     else: #These are the predetermined initialization cases
309         vals = np.zeros((a,b,c,d))
310         if itype == -1: #Initial guess is the moments obtained from the poscar file
311             for i in r:
312                 vals[i] = ms[i]
313             print("Initializing the exact value given in exchange.out")
314         if itype == -2: #try the literature orientation ONLY WORKS FOR CELLS OF 0
            ↪ and bigger
315             for i in r:
316                 for j in range(b):
317                     for k in range(c):
318                         for l in range(d):
319                             if (j + k + l)%2 == 0:
320                                 if i < 3:
321                                     vals[i,j,k,l] = abs(ms[i])
322                                 else:
323                                     vals[i,j,k,l] = -abs(ms[i])
324                             else:
325                                 if i < 3:
326                                     vals[i,j,k,l] = -abs(ms[i])
327                                 else:
328                                     vals[i,j,k,l] = abs(ms[i])
329             print("Initializing the literature configuration")
330     return vals
331
332 def initialize_vars_2(ranges,m,unique_ats,la, moments, init = 0, boost = False,
    ↪ starts = {}):
```

```
333     #ranges include the cells that we want to sum over, already including the 0 if
        ↪ necessary
334     #m is the model
335     #unique ats is the unique atoms in the unit cell
336     #la is the structure object
337     s = ' '
338     if boost == False:
339         vals = initial_vals(ranges,unique_ats,moments,init)
340     else:
341         print("Using previous ending")
342     at = [[] for i in range(len(unique_ats))]
343     x_names = [[] for i in range(len(unique_ats))]
344     for e in range(len(unique_ats)):
345         at[e] = list_3D(len(ranges[0]), len(ranges[1]), len(ranges[2]), None)
346         x_names[e] = list_3D(len(ranges[0]), len(ranges[1]), len(ranges[2]), '')
347     for h in range(len(unique_ats)):
348         for i in ranges[0]:
349             for j in ranges[1]:
350                 for k in ranges[2]: #so these all walk from negative to positive
                    ↪ values potentially. OUT OF RANGE ERROR -> wrap
351                     try:
352                         x_names[h][i][j][k] = f'{unique_ats[h]}q{i}q{j}q{k}'
353                         bound = abs(moments.get(unique_ats[h]))
354                         if boost == False:
355                             val = vals[h,i,j,k]
356                         else:
357                             if len(starts) == 0:
358                                 print("ERROR: No data found for second run")
359                             #print(str(unique_ats[h])+s+str(i)+s+str(j)+s+str(k))
360                             val =
                                ↪ starts.get(str(unique_ats[h])+s+str(i)+s+str(j)+s+str(k))
361                         at[h][i][j][k] = m.Var(lb = -bound, ub = bound, value =
                            ↪ val, name = x_names[h][i][j][k])
362                     except IndexError:
363                         print("ERROR: 0 has to be part of all the ranges!!!")
364     #print("Done with initializing vars")
        ↪
365     return at, x_names
366
367 #=====================================================================#
368 #                        DEFINING OPTIMIZATION                        #
369 #=====================================================================#
370
371 def J_sum_2(m,J_d,q,var,star,stop,i, typ = 0, k = 0, lim = 0, ori = [1,1,1,0]):
    ↪ #main function for the objective function
372
373     s = ' '
374     # for j in range(stop-star):
375     #     if J_d.get(q[i][3] +s+ q[j][3] +s+ str(int(q[i][0]-q[j][0])) +s+
        ↪ str(int(q[i][1]-q[j][1])) +s+ str(int(q[i][2]-q[j][2]))) == None:
376     #         print(q[i][3] +s+ q[j][3] +s+ str(int(q[i][0]-q[j][0])) +s+
        ↪ str(int(q[i][1]-q[j][1])) +s+ str(int(q[i][2]-q[j][2])))
377     # J indices
378     #print(J_d.get(temp[0]))
379     #print(type(J_d.get(temp[0])))
380     if typ == 1:
381         temp = [(q[k][3] +s+ q[j][3] +s+ str(int(q[k][0]-q[j][0])) +s+
            ↪ str(int(q[k][1]-q[j][1])) +s+ str(int(q[k][2]-q[j][2]))) for j in
            ↪ range(star,stop)]
382         #print(temp)
```

```
383         length = 0
384         l = star
385 #         print('max: ' +str(stop))
386         while length < lim:
387             #print('iteration: '+str(l))
388             if type(J_d.get(temp[l-star])) != None and (int(q[k][0]-q[l][0]) != 0
                ↪  or int(q[k][1]-q[l][1])  != 0 or int(q[k][2]-q[l][2]) != 0):
389                 length += 1
390             #if J_d.get(temp[l-star]) != None:
391                 #print(type(J_d.get(temp[l-star])))
392             l += 1
393             if l == stop:
394                 break
395         return m.Minimize(-2*sum(J_d.get(temp[j-star])*var[i]*var[j] for j in
            ↪  range(star,l) if J_d.get(temp[j-star]) is not None and
            ↪  (J_sum_cond([int(q[k][0]-q[j][0]),
            ↪  int(q[k][1]-q[j][1]),int(q[k][2]-q[j][2])],ori = ori)))), l
396     if typ == 0:
397         temp = [(q[i][3] +s+ q[j][3] +s+ str(int(q[i][0]-q[j][0])) +s+
            ↪  str(int(q[i][1]-q[j][1])) +s+ str(int(q[i][2]-q[j][2]))) for j in
            ↪  range(star,stop)]
398         return m.Minimize(-2*sum(J_d.get(temp[j-star])*var[i]*var[j] for j in
            ↪  range(star,stop) if J_d.get(temp[j-star]) is not None))
399
400
401
402 def optimize(J_matrix,ranges, path, atom_names, latt, moments, J_reach, solver = 1,
    ↪  m_name = "test", info = 1, init_ = 0, sens = False, boost = False, starts =
    ↪  {}):
403     m = GEKKO(name = m_name, remote = False)
404     ran = [None, None, None]
405     inc = 0
406
407     for i in range(3): #Check if we need another value due to including 0.
408         if np.sign(ranges[2*i+1]) != np.sign(ranges[2*i]):
409             inc = 1
410         ran[i] = range(int(ranges[2*i]),int(ranges[2*i+1] + inc)) #makes ranges to
            ↪  sum over
411
412     unique_atoms = np.unique(atom_names)
413     atom_range = range(len(unique_atoms))
414     if boost == False:
415         atoms, names = initialize_vars_2(ran, m, unique_atoms,latt,moments,init =
            ↪  init_)
416     else:
417         atoms, names = initialize_vars_2(ran, m, unique_atoms,latt,moments,init =
            ↪  init_, boost = boost, starts = starts)
418     atoms_flat, at_index = flatten_matrix(atoms,ran, atom_range, unique_atoms)
419     cons = periodic_bc_2(m, J_matrix, unique_atoms, atoms_flat, at_index, ranges,
        ↪  J_reach)
420     #J_par = initialize_pars(m, J_matrix)
421     test = obj_fct_3(J_matrix,atoms_flat,at_index, m)
422     p = os.path.join(os.getcwd(),path)
423     m._path = p
424     m.options.WEB = 0
425     m.options.solver = solver
426     if sens == True:
427         m.options.SENSITIVITY = 1
428     #m.options.CSV_WRITE = info
429     print("Starting optimization")
```

```
430     #m.solve(disp=True)
431     with open(path/'iterations', 'w') as f:
432         with redirect_stdout(f):
433             m.solve(disp=True)
434     print("Finished optimization")
435
436
437
438 def obj_fct_3(J_mat, var_vec, q, m): # makes the main objective function of the
    ↪  system
439     cutoff = 200
440     eq_num = count(len(var_vec),cutoff)
441     obj_fct = [None for j in range(int(eq_num))]
442     k = 1
443     for i in range(1,len(var_vec)):
444         if var_vec[i] is None:
445             continue
446         if i > cutoff:
447             eqs = int(np.floor(i/cutoff)+1)
448             length_eq = int(i - cutoff*(eqs-1))
449             obj_fct[i] = J_sum_2(m,J_mat,q,var_vec,0,cutoff,i)
450             for l in range(1,eqs):
451                 if l == eqs-1:
452                     obj_fct[-k] = J_sum_2(m,J_mat,q,var_vec,l*cutoff,l*cutoff +
                        ↪  length_eq,i)
453                 else:
454                     obj_fct[-k] =
                        ↪  J_sum_2(m,J_mat,q,var_vec,l*cutoff,(l+1)*cutoff,i)
455                 k += 1
456         else:
457             obj_fct[i] = J_sum_2(m,J_mat,q,var_vec,0,i,i)
458     print("Done with making obj function")
459     return obj_fct[obj_fct != 0]
460
461 #=========================================================================#
462 #                          SMALL HELP FUNCTIONS                           #
463 #=========================================================================#
464
465 def check_non_mag_atoms(at_dict):
466     lst = [[],[]]
467     threshhold = 0.1
468     for i in at_dict:
469         g = re.sub(' \d','',i)
470         if abs(at_dict[i]) < threshhold:
471             lst[1].append(g)
472         else:
473             lst[0].append(g)
474
475     return [list(set(lst[0])),list(set(lst[1]))]
476
477 def cut_equation(m, J_mat, uni_at, var_vec, ind, J_reach, cell,star, stop, i, typ =
    ↪  0, j = 0, ori = [1,1,1,0]): #go here when we've decided upon an equation
478     # GEKKO has a max line length of 15000 characters; we prevent going over that
        ↪  amount here.
479     # i is the var of interest
480     #j is its equivalent brother
481     #how many equations do we reall want? We want to start summing over everything
482     cutoff = 200
483     size =
        ↪  len(uni_at)*min(J_reach[0],cell[0])*min(J_reach[1],cell[1])*min(J_reach[2],cell[2])
        ↪  #approximate num of atoms in equations
```

```
484     eqs = int(np.floor(size/cutoff)+1)
485     fct = [None for k in range(eqs)]
486     for k in range(eqs):
487         if k == eqs -1:
488             fct[k], star = J_sum_2(m, J_mat, ind, var_vec, star, stop, i,typ = 1, k
            ↪  = j, lim = size/eqs, ori = ori)
489         else:
490             fct[k], star = J_sum_2(m, J_mat, ind, var_vec, star, stop, i,typ = 1, k
            ↪  = j, lim = size/eqs, ori = ori)
491     return fct
492
493 def list_3D(len_x,len_y,len_z,ty):
494     lst = [[ [ty for i in range(len_x)] for j in range(len_y)] for k in
        ↪  range(len_z)]
495     return lst
496
497 def check_line_length(filename): #To check if we go over the maximum length of one
    ↪  line in the optimization file.
498     filename = Path(str(filename) +".apm")
499     with open(filename) as f:
500         lines = f.readlines()
501     lengths = np.zeros(len(lines))
502     for i in range(len(lines)):
503         lengths[i] = len(lines[i])
504     return max(lengths)
505
506 def initialize_pars(m,J_matrix): #giving each parameter a name; might only be semi
    ↪  usefull.
507     J_pars = {}
508     for i in J_matrix:
509         if J_pars.get(i) != None: #checking if we have extra names or not
510             print(i)
511         J_pars[i] = m.Param(value = J_matrix.get(i))
512     return J_pars
513
514 def count(var,cutoff): #function that also works in the line cut function to make
    ↪  sure we don't run out of space
515     tot_eq = 0
516     for i in range(int(np.floor(var/cutoff)+1)):
517         if i < np.floor(var/cutoff):
518             tot_eq += cutoff*(i+1)
519         else:
520             tot_eq += (var - cutoff*i)*(i+1)
521     return int(tot_eq)
522
523 def find_lattice_pars(im,lat): #adds the actuals lattice paramters to the J matrix.
524     J_ext = np.copy(im)
525     J_ = np.concatenate((J_ext,np.zeros((len(J_ext[:,0]),3))),axis = 1)
526     x, y, z = range(4,7)
527     c, b, a = range(-3,0)
528     for i in range(len(J_[:,0])):
529         J_[i,c:] = lat.lattice.inv_matrix.dot(J_[i,x:(z+1)])
530     return J_
531
532 #Second try; flattens the matrix and accounts for double counting
533 def flatten_matrix(mat, ra,at_ra,nam):
534     #mat is the matrix, ra are the ranges, at_ra is the atom range and nam are the
        ↪  unique atom names
535     lst = []
536
```

```
537     indices = [[] for i in
        ↪  range(len(mat[0][0][0])*len(at_ra)*len(mat[0])*len(mat[0][0]))]
538     for w in at_ra:
539         for t in ra[0]:
540             for u in ra[1]:
541                 for v in ra[2]:
542                     lst.append(mat[w][t][u][v])
543                     ind = len(lst) -1
544                     indices[ind] = [t, u, v, nam[w]]
545     #print(len(lst))
546     #print(indices[0:10])
547     #print("Done with flattening variable matrix")
548     return lst, indices
549
550 def find_unit_cell(coords, new_names, struct_obj, which = 0,  bounds =
    ↪  np.array([0,0,0,1,1,1])): #selects a smaller subset to print
551     '''
552     We assume here that coords are in the shape of lattice vectors (!!)
553     '''
554     needed_co = []
555     if which == 0:
556         #print(coords[0,0:3])
557         for i in range(len(coords[:,0])):
558             if np.all(coords[i,0:3] < bounds[[1,3,5]]) and np.all(coords[i,0:3] >=
                ↪  bounds[[0,2,4]]):
559                     needed_co.append(i)
560         coords = coords[needed_co,:]
561         names = np.array(new_names)
562         names = names[needed_co]
563     return coords, names
564
565 def calculate_tot_mom(spins,out_fol,c,modn): #as mentioned; sums over all magnetic
    ↪  moments to find the moment per unit cell
566     tot_mom = sum(spins[:,-1])
567     off = np.zeros(3)
568     for i in range(3):
569         if np.sign(c[2*i+1]) != np.sign(c[2*i]):#WE actually always know this as we
            ↪  require 0 to be part of the range!
570             off[i] = 1 #Still, just to make sure though.
571     av_mom = tot_mom/((c[1]-c[0]+off[0])*(c[3]-c[2]+off[1])*(c[5]-c[4]+off[2]))
572     with open(out_fol/'mag_moment', 'w') as f:
573         with redirect_stdout(f):
574             print('Total magnetic moment: ' + str(tot_mom))
575             print('Magnetic moment per unit cell: ' + str(av_mom))
576             print('Max line length in model file: ' + str(check_line_length(out_fol
                ↪  / modn)))
577     print('Total magnetic moment: ' + str(tot_mom))
578     print('Magnetic moment per unit cell:' + str(av_mom))
579     print()
580     return
581
582 #=====================================================================#
583 #                         MAIN RUNNER FUNCTIONS                       #
584 #=====================================================================#
585
586 def used_data(in_fol,out_fol,call_type = 0): #functions that ties all the others
    ↪  together
587     dir_path = os.path.dirname(os.path.realpath(__file__))
588     os.chdir(dir_path)
589     file = in_fol / "POSCAR"
```

```
    TB2J = True
    if os.path.exists(in_fol / "exchange.out"):
        J_file = in_fol / "exchange_short.out"
        J_old = in_fol / "exchange.out"
        mode = 0
        hor_vals = 10
    else:
        TB2J = False
        J_file = in_fol / "jdata"
        mode = -1 #to read the data from jdata
        hor_vals = 6
    if call_type == 0:
        try:
            os.mkdir(out_fol)
        except OSError:
            print("Warning!! Out folder already exists")
            # cont = input('Continue anyway? [y/n]')
            # if cont == 'n':
            #     return None
        if os.path.exists(J_file) == False and TB2J == True:
            J_short(J_old,J_file)
    if TB2J:
        at_mom = read_moments(J_old)
        at_conv_names = None

    else:
        at_mom, at_conv_names = read_moments(in_fol / "moments_in", TB2J = TB2J)

    mag_ats = check_non_mag_atoms(at_mom)

    lat = Poscar.from_file(file).structure
    lat.remove_species(mag_ats[1])

    sec_lat = Poscar.from_file(file).structure
    sec_lat.remove_species(mag_ats[0])
    Js, at_names = read_file(J_file,hor_vals, mode, convert_names=at_conv_names)

    biggest_dist = np.zeros(3)
    if TB2J:
        J_better = find_lattice_pars(Js, lat)
        for i in range(3):
            biggest_dist[i] = max(J_better[:,-3+i])
    else:
        J_better = Js
        for i in range(3):
            biggest_dist[i] = max(J_better[:,i])
    J_dlvec, sites_dic = J_as_dict(J_better, at_names,lat, 3)
    #print(type(J_dlvec.get("Mn 1 Mn 4 2 1 1")))
    return lat, sec_lat, at_names, at_mom, J_dlvec, sites_dic, dir_path,
    ↪  biggest_dist

def full_opt(in_fol,out_fol,calc_range, modn = 'mod',title_ = "First run", initi =
↪  0, sensitivity = False,plot = True, plot_range = np.array([0,1,0,1,0,1]), boost
↪  = False, starts = {}):
    '''
    Parameters
    ----------
    in_fol : Path
        Folder that contains the input (POSCAR and exchange.out) files of material
    out_fol : Path()
```

```
647         Folder to print output files to
648     calc_range : list
649         1D list that contains the unit cells to concider in the calculation in the
    ↪   form [a_min,a_max,b_min,b_max,c_min,c_max]
650     plot_range : numpy array
651         1D array that contains the unit cells to plot in the form
    ↪   [a_min,a_max,b_min,b_max,c_min,c_max]
652     modn : str, optional
653         The name for the model. The default is 'mod'.
654     title_ : str, optional
655         Title for the plot. The default is "First run".
656     initi : int
657         Type of initialization.
658         0 = any val within bounds
659         1 = only positive values
660         2 = sign equal to sign in poscar
661 
662     Returns
663     -------
664     None.
665 
666     '''
667     # for i in np.unique(at_names):
668     #     J_dlvec[i + ' '+i+' 0 0 0'] = 0
669     lat, sec_lat, at_names, at_mom, J_dlvec, sites_dic, dir_path, J_range =
    ↪   used_data(in_fol,out_fol)
670     #print(J_dlvec)
671     if lat == None:
672         print('Stopping run: wrong folder')
673         return
674     if len(re.findall('I',str(out_fol))) > len(re.findall('I',str(in_fol))):
    ↪   #selecting which solver to use based on the folder name (should contain a I
    ↪   for IPOPT and A for APOPT)
675         solve = 3
676         print("Using the IPOPT Solver") #Both A and I present defaults to IPOPT
677     elif len(re.findall('A',str(out_fol))) > len(re.findall('A', str(in_fol))):
678         solve = 1
679         print("Using the APOPT Solver")
680     else:
681         solve = 3
682     optimize(J_dlvec,calc_range,dir_path / out_fol,at_names,lat,at_mom, J_range,
    ↪   solver = solve, m_name = modn, init_ = initi, sens = sensitivity, boost =
    ↪   boost, starts = starts)
683 
684     spin_states, spin_names = read_file(out_fol / 'results.json',5,1)
685     calculate_tot_mom(spin_states,out_fol,calc_range,modn)
686     if plot == True:
687         plotgrid,plot_names = find_unit_cell(spin_states, spin_names, lat, bounds =
    ↪       plot_range)
688         cols = ['gray','mediumaquamarine','lightcoral']
689     #    plot_non_mag(plotgrid[:,0:3],sec_lat, show = False)
690         mp.plot_scatter(plotgrid[:,0:3],plotgrid[:,3],plot_names,sites_dic, cols,
    ↪       title_, latt = 1, struct_obj = lat, obj2 = sec_lat, print_lat = True,
    ↪       spins= True,point_scale=100)
691 
692 #=====================================================================#
693 #                        MULTIPLE RUNS CODE                           #
694 #=====================================================================#
695 
696 def run_times(in_fol,start_range,runs,inits, sensi = False, title = 'First run'):
    ↪   #simple loop to run the code a said number of times
```

```
    s = '_' #also automatically makes a foldername
    k = 0
    for i in runs:
        for j in inits:
            out_fol = in_fol
            ↪ /('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j))
            full_opt(in_fol,out_fol, start_range, initi= j,plot = False,
            ↪ plot_range=np.array([-1,3,-1,3,-1,3]), sensitivity=sensi)
            k += 1
            print('Done with run number ' + str(k))

def run_thermo(in_fol,start_range,runs,inits,title = 'First run concatenated',
↪ plt_range = np.array([0,2,0,2,0,2]), mod = [0], key_1 = 'r', key_2 = '\d',
↪ change = 0.6):
    s = '_'
    t = ' '
    k = 0
    for i in runs:
        for j in inits:
            start_time = time.time()
            new_start = {}
            out_fol = in_fol
            ↪ /('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j))
            full_opt(in_fol,out_fol, start_range, initi= j,plot = False,
            ↪ plot_range=np.array([-1,3,-1,3,-1,3]))
            spins, names = read_file(out_fol / 'results.json', 5, 1)
            if mod[i] >= 0:
                spins[:,3] = boost_rand(spins[:,3], mode = mod[k])
            print("First run done; giving progam a kick")
            #print(np.unique(names))
            for l in range(len(names)):
                new_start[names[l]+t+ str(int(spins[l,0]))+t+str(int(spins[l,1]))
                ↪ +t+str(int(spins[l,2]))] = spins[l,3]
            if mod[i] >= 0:
                out_fol_2 = in_fol
                ↪ /('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j)+s)
                full_opt(in_fol,out_fol_2, start_range, initi= j,plot = False,
                ↪ plot_range=np.array([-1,3,-1,3,-1,3]),boost = True, starts =
                ↪ new_start)
            #print(new_start)
            if mod[i] < 0:
                new_starts = boost_sym(new_start, spins, names, change = change)
                while new_starts[0].get('rerun') == True: #this does mean that you
                ↪ can get really unlucky and be trapped in a loop.
                    print('Starting solution is concidered too poor; running
                    ↪ program again')
                    full_opt(in_fol,out_fol, start_range, initi= j,plot = False,
                    ↪ plot_range=np.array([-1,3,-1,3,-1,3]))
                    spins, names = read_file(out_fol / 'results.json', 5, 1)
                    new_start = {}
                    for l in range(len(names)):
                        new_start[names[l]+t+
                        ↪ str(int(spins[l,0]))+t+str(int(spins[l,1]))
                        ↪ +t+str(int(spins[l,2]))] = spins[l,3]
                    new_starts = boost_sym(new_start, spins, names, change =
                    ↪ change)

                if len(new_starts[0]) > 0:
                    print(f'Running {len(new_starts)} symmetric version(s) of
                    ↪ previous outcome')
```

```
                    n = ['a','b','c','d','e','f']
                    for m in range(len(new_starts)):
                        out_fol_3 = out_fol_2 = in_fol
                        ↪ /('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j)+n[m])
                        full_opt(in_fol,out_fol_3, start_range, initi= j,plot =
                        ↪ False,boost = True, starts = new_starts[m])
                else:
                    print('current solution is considered good enough symmetry
                    ↪ wise')
            k += 1
            run_time = time.time() - start_time
            minutes = int(run_time/60)
            seconds = np.round((run_time/60 - minutes)*60,2)
            with open(in_fol/'runtimes','a') as f:
                with redirect_stdout(f):

                    ↪ print('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j)+'
                    ↪ '+str(run_time))
            print('rI'+str(start_range[1]-start_range[0]+1)+s+str(i)+s+str(j)+'
            ↪ done with a runtime of '+str(minutes)+':'+str(seconds)+' minutes')
    opt_sol_finder(in_fol, key_1, key_2)

def opt_sol_finder(in_fol, key_1, key_2, cubic = True, copy = True, prnt = True,
↪ mode = 1):
    finals = []
    names = []
    in_fol = Path(in_fol)
    for fn in sorted(os.listdir(in_fol)):
        if os.path.isdir(in_fol/fn) and len(re.findall(key_1,str(fn))) > 0 and
        ↪ len(re.findall(key_2,str(fn))) > 0:
            temp_fol = in_fol / fn
            if str(fn)[-1].isnumeric():
                first_run = True
            else:
                first_run = False
            if len(re.findall('A', fn)) > 0: #Old colors for other method (APOPT
            ↪ instead of IPOPT)
                ft = [1,2] #selects which mode to use of the read function
            else:
                ft = [3,3]
            J_short(temp_fol/'iterations',temp_fol/'iter',ftype=ft[0])
            ite = read_file(temp_fol/'iter',10,ft[1])
            if cubic == True:
                ite[:,1] = ite[:,1]/(int(re.findall('\d',str(fn))[0])**3) #this is
                ↪ not accurate probably because of extra 'ghost' interactions
            moments = read_file(temp_fol/'mag_moment',2,4)
            finals.append([str(fn),ite[-1,1], ite[-1,0], moments[1,1]])
            if mode == 1:
                if not first_run:
                    if str(fn)[-5].isnumeric(): #check if run index has 2 numbers
                    ↪ or one
                        names.append(str(fn)[-5:-1])
                    else:
                        names.append(str(fn)[-4:-1])
                else:
                    if str(fn)[-4].isnumeric():
                        names.append(str(fn)[-4:])
                    else:
                        names.append(str(fn)[-3:])
            if mode == 2:
```

```
                if str(fn)[-1].isnumeric() == False:
                    names.append(str(fn)[:-1])

    with open(in_fol / "results.txt", 'w') as f:
        for i in range(len(finals)):
            f.write(finals[i][0] + ' ' + str(finals[i][1]) +' '+
            ↪  str(finals[i][2])+' '+str(finals[i][3])+' '+str(names[i])+'\n')
    nam_arr = np.array(names)
    #print(nam_arr)
    fin_arr = np.array(finals)
    runs = np.unique(nam_arr, return_inverse = True, return_counts = True)
    #print(runs[1])
    #print(runs[0])
    #print(nam_arr)
    if copy:
        for i in range(len(runs[0])):
            temp_arr = fin_arr[(runs[1]==i),:]
            #print(temp_arr[:,1])
            temp_win = np.argmin(temp_arr[:,1].astype(float))
            n_t = temp_arr[temp_win,0]
            #print(n_t)
            if prnt:
                print(str(n_t)+' is the optimal result; copying to f folder')

            shutil.copytree(in_fol/str(n_t), in_fol/f'f{str(n_t)[1:-1]}')

#=====================================================================#
#                        ITERATING FUNCTIONS                          #
#=====================================================================#

def boost_sym(d, s, n, change = 0.6): #d = dictionary, s = spins, n = names
    #we again assume that 0 is within each range.
    un = np.unique(n)
    minmax = np.zeros(6)
    ranges = [None,None,None]
    for i in range(3):
        minmax[2*i] = min(s[:,i])
        minmax[2*i+1] = max(s[:,i])

        ranges[i] = range(int(minmax[2*i]),int(minmax[2*i+1]+1))
    sym_array = np.zeros((len(ranges[0])*len(ranges[1])*len(ranges[2]),len(un)+3))
    m = 0
    for i in ranges[0]:
        for j in ranges[1]:
            for k in ranges[2]: #we sum over each unit cell and evaluate signs
                temp = np.zeros(len(un)+3)
                for l in range(len(un)):
                   temp[l] = np.sign(d.get(un[l]+' '+str(i)+' '+str(j)+' '+str(k)))
                #temp = [1,-1,1,-1] each place corresponding to the sign of the
                ↪  spin of an atom.
                temp[-3] = i
                temp[-2] = j
                temp[-1] = k
                sym_array[m] = temp
                m+= 1
    tot_length = len(sym_array[:,0]) #amount atoms
    #print(tot_length) #should be equal to 32 for a 2x2x2 MnO cell
    #print(sym_array)
    abs_sym = sym_array.copy()
    for i in range(len(abs_sym[:,0])):
```

```
847             abs_sym[i,:-3] = abs_sym[i,:-3]*abs_sym[i,0]
848         configs = np.unique(abs_sym[:,:-3], return_inverse = True, return_counts=True,
            ↪  axis = 0)
849         ind = []
850         # print('first: '+ str(configs[0]))
851         # print(configs[1])
852         # print(configs[2])
853         for i in range(len(configs[2])):
854             if configs[2][i] > tot_length/3: #we see this configuration as significant
                ↪  as its at least 30% of the material
855                 ind.append(i)
856         if len(ind) > 0:
857             ret_dicts = [{} for i in range(len(ind))]
858         else:
859             ret_dicts = [{'rerun': True}] #we have a very antisymmetric solution. We
                ↪  try to run the program again with a random initialization
860             return ret_dicts
861 
862         p = 0
863         for i in ind: #sum over all the prominent configurations
864             perfect = 0
865             old_ind = np.where(configs[1] == i)[0] #indices in array where its brothers
                ↪  are present
866             cell_sym = np.zeros((len(ranges[0]),len(ranges[1]),len(ranges[2])))
867             for a in old_ind: #what does this do exactly?
868                 if np.all(abs_sym[a] == sym_array[a]):
869                     cell_sym[int(sym_array[a,-3]), int(sym_array[a,-2]),
                        ↪  int(sym_array[a,-1])] = 1
870                 else:
871                     cell_sym[int(sym_array[a,-3]), int(sym_array[a,-2]),
                        ↪  int(sym_array[a,-1])] = -1
872             #now the matrix is set up so we can start looking for symmetries
873             # we'll look in each direction and check each column for its dominant
                ↪  pattern
874             # print(abs_sym)
875             # print(sym_array)
876             #print(cell_sym)
877             if len(np.where(cell_sym == 0)[0]) == 0:
878                 perfect += 1 #we only have one type of unit cell -> bonus points :)
879             cs = np.zeros(3).astype(int)
880             pattern = np.zeros(3).astype(int)
881             for coord in range(3):
882                 ferro = 0
883                 ant_ferro = 0 #these names are misleading its more like 1 - cell
                    ↪  symmetric and 2 - cell symmetric.
884                 for cs[coord-1] in ranges[coord-1]:
885                     for cs[coord-2] in ranges[coord-2]:
886                         for cs[coord] in range(int(minmax[2*coord]),int(minmax[2*coord
                            ↪  + 1])):
887                             ad = np.zeros(3).astype(int)
888                             ad[coord] = 1
889                             if cell_sym[cs[0],cs[1],cs[2]] ==
                                ↪  cell_sym[cs[0]+ad[0],cs[1]+ad[1],cs[2]+ad[2]] and
                                ↪  cell_sym[cs[0],cs[1],cs[2]] != 0:
890                                 ferro += 1
891                             elif cell_sym[cs[0],cs[1],cs[2]] ==
                                ↪  -cell_sym[cs[0]+ad[0],cs[1]+ad[1],cs[2]+ad[2]] and
                                ↪  cell_sym[cs[0],cs[1],cs[2]] != 0:
892                                 ant_ferro += 1
893 
```

```
            pattern[coord] = np.sign(ferro-ant_ferro) #-1: switching, 0 equal and 1 is ferro
        # print(pattern)
        unaltered = True
        if perfect == 1 and sum(pattern) == 1 and np.all(pattern != 0):
            print('Solution looks good; trying other symmetric configuration to check optimallity')
            pattern = [1,1,1] #try special different solution as
            unaltered = False

        if perfect == 1 and sum(pattern) == -1 and np.all(pattern != 0):
            pattern = [-1,-1,-1]
            print('Solution looks good; trying other symmetric configuration to check optimallity')
            unaltered = False #we do this now for very neat solutions that would otherwise be considered
        #we now have a list with the prominent patterns per direction.
        #we only take the most likely one otherwise the amount of runs grows exponentially as we're also checking different very likely layouts
        #so [1 1 0] -> [1 1 1] and [1 -1 0] -> [1 -1 1] as we will guess that 1 cell symmetry is more likely
        # only when [-1 -1 1]
        # When are we satisfied??? [1 1 1], [-1 -1 -1]?

        if np.all(pattern == pattern[0]) and unaltered:
            check = True
            for j in ranges[0]:
                for k in ranges[1]:
                    for l in ranges[2]:
                        if pattern[0]**(j+k+l)*cell_sym[0,0,0] != cell_sym[j,k,l] and cell_sym[0,0,0] != 0:
                            check = False
            #This works for our purposes, however, for larger materials this can still return a false positive.
            #Hence should be altered to check each unit cell for the pattern maybe, or a condition should be built in
            #that checks for irregularities
            if check:
                perfect += 1

        if perfect == 2: #we are satisfied with the current result and return the
            #print(pattern)
            #print(sym_array)
            return ret_dicts #we don't really do anything here as we assume the solution to be optimal
        for j in range(3):
            if pattern[j] == 0:
                pattern[j] = 1

        if cell_sym[0,0,0] == -1:
            cell_sym = cell_sym*(-1) #to make sure that we dont switch/alter too many atoms; as -1 and 1 are similar solutions
        new_cells = np.zeros((len(ranges[0]),len(ranges[1]),len(ranges[2])))

        for j in ranges[0]:
            for k in ranges[1]:
                for l in ranges[2]:
                    val = pattern[0]**j*pattern[1]**k*pattern[2]**l
                    c = 1
                    if cell_sym[j,k,l] != val:
```

```
                        c = change
                    new_cells[j,k,l] = c*val
                    #print(new_cells)
                    for m in range(len(un)):
                        ret_dicts[p][un[m]+' '+str(j)+' '+str(k)+' '+str(l)] =
                        ↪ c*val*abs(d.get(un[m]+' '+str(j)+' '+str(k)+'
                        ↪ '+str(l)))
        #now we have a first run; if our symmetry is not 'perfect' in each
        ↪ direction; we also try a second run to check if our solution is really
        ↪ optimal
        if abs(sum(pattern)) != 3:
            if sum(pattern) > 0:
                new_pat = [1,1,1] #meaning that if we have -1, 1, 1; we also try 1
                ↪ 1 1
            else:
                new_pat = [-1,-1,-1] #if we have -1 -1 1, also try -1, -1, -1.
                #NOTE that we disregard the other configurations (if we have -1 1 1
                ↪ we don't try 1 -1 1 and 1 1 -1)

            new_cells_2 = np.zeros((len(ranges[0]),len(ranges[1]),len(ranges[2])))
            ret_dicts.append({})
            for j in ranges[0]:
                for k in ranges[1]:
                    for l in ranges[2]:
                        val = new_pat[0]**j*new_pat[1]**k*new_pat[2]**l
                        c = 1
                        if cell_sym[j,k,l] != val:
                            c = change
                        new_cells_2[j,k,l] = c*val
                        #print(new_cells)
                        for m in range(len(un)):
                            ret_dicts[-1][un[m]+' '+str(j)+' '+str(k)+' '+str(l)] =
                            ↪ c*val*abs(d.get(un[m]+' '+str(j)+' '+str(k)+'
                            ↪ '+str(l)))
        p += 1
    #print(ret_dicts[0])
    #print(new_cells)
    return ret_dicts

def boost_rand(vec, mode): #previous implementation of rerun method
    if mode == 0: #randomly select a couple of atoms to give a kick
        rng = np.random.default_rng()
        ind = rng.choice(len(vec),int(len(vec)/3),replace = False)
        for i in ind:
            vec[i] = vec[i] - vec[i]*0.6
    if mode == 1:
        add = np.random.rand(len(vec))
        vec = vec - add*vec*0.2
    if mode == 2:
        vec = vec - vec*0.05
        rng = np.random.default_rng()
        ind = rng.choice(len(vec),int(len(vec)/8),replace = False)
        for i in ind:
            vec[i] = -vec[i]
    if mode == 3:
        rng = np.random.default_rng()
        ind = rng.choice(len(vec),int(len(vec)/3),replace = False)
        for i in ind:
            vec[i] = vec[i] - vec[i]*1.5
    return vec
```

```
995
996 def test_boosts(in_fol, start_fol, start_range,title = 'First test', mod = -1,
    ↪  change = 0.6): #testing our change function on an existing suboptimal solution
997     start_time = time.time()
998     t = ' '
999     spins, names = read_file(in_fol / start_fol / 'results.json', 5, 1)
1000    new_start = {}
1001    for l in range(len(names)):
1002        new_start[names[l]+t+ str(int(spins[l,0]))+t+str(int(spins[l,1]))
            ↪  +t+str(int(spins[l,2]))] = spins[l,3]
1003    if mod < 0:
1004        new_starts = boost_sym(new_start, spins, names, change = change)
1005        if new_starts[0].get('rerun') == True: #this does mean that you can get
            ↪  really unlucky and be trapped in a loop.
1006            print('Starting solution is concidered too poor; we skip this run as it
                ↪  is not interesting')
1007            return
1008
1009        if len(new_starts[0]) > 0:
1010            print(f'Running {len(new_starts)} symmetric version(s) of previous
                ↪  outcome')
1011            n = ['a','b','c','d','e','f']
1012            for m in range(len(new_starts)):
1013                out_fol_3 = in_fol /(re.sub('I',str(change)[-1], str(start_fol))
                    ↪  +n[m])
1014                full_opt(in_fol,out_fol_3, start_range, initi= 0,plot = False,boost
                    ↪  = True, starts = new_starts[m])
1015        else:
1016            print('current solution is considered good enough symmetry wise')
1017    run_time = time.time() - start_time
1018    minutes = int(run_time/60)
1019    seconds = np.round((run_time/60 - minutes)*60,2)
1020    with open(in_fol/'runtimes','a') as f:
1021        with redirect_stdout(f):
1022            print(re.sub('I',str(change)[-1], str(start_fol))+' '+str(run_time))
1023    print(re.sub('I',str(change)[-1], str(start_fol))+' done with a runtime of
        ↪  '+str(minutes)+':'+str(seconds)+' minutes')
1024
1025
1026 def main():
1027     return
1028
1029 if __name__ == '__main__':
1030     main()
```