Partn: A Plugin Implementation of the Activation Relaxation Technique Nouveau Hijacking a Minimisation Algorithm

Matic Poberžnik, Miha Gunde, Nicolas Salles, Antoine Jay, Anne Hémeryck, Nicolas Richard, Normand Mousseau, Layla Martin-Samos

▶ **To cite this version:**

Matic Poberžnik, Miha Gunde, Nicolas Salles, Antoine Jay, Anne Hémeryck, et al.. Partn: A Plugin Implementation of the Activation Relaxation Technique Nouveau Hijacking a Minimisation Algorithm. Computer Physics Communications, In press, ⟨10.2139/ssrn.4360939⟩. ⟨hal-04238051⟩

HAL Id: hal-04238051

https://laas.hal.science/hal-04238051v1

Submitted on 11 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

pARTn: a plugin implementation of the Activation Relaxation Technique nouveau that takes over the FIRE minimisation algorithm

M. Poberznik[a], M. Gunde[e], N. Salles[a], A. Jay[b], A. Hemeryck[b], N. Richard[c], N. Mousseau[d], L. Martin-Samos[a]

[a] *CNR-IOM/Democritos National Simulation Center, Istituto Officina dei Materiali, c/o SISSA, via Bonomea 265, IT-34136 Trieste, Italy*
[b] *LAAS-CNRS, Université de Toulouse, CNRS, Toulouse, France*
[c] *CEA, DAM, DIF, F-91297 Arpajon, France*
[d] *Université de Montréal, Canada*
[e] *Institute Ruđer Bošković, Bijenička 54, 10000 Zagreb, Croatia*

Abstract

Nowadays, the interoperability and interfacing of codes and libraries have become crucial aspects of software development and engineering, and the basis for enabling and simplifying the sharing of methods and tools, both within and among communities. One of the most important bottlenecks that arises when developing and maintaining an interface of a library with an already existing software, is to keep it aligned with the development route of the latter. This might include significant changes, such as changes in the data structures used by the library, which are communicated through the interface.

In this paper, an approach for inserting a new algorithm into existing software is presented, through a minimally invasive interface, that takes over an already present algorithm, and thus changes its original purpose. The approach is applied to the well-established Activation-Relaxation Technique nouveau (ARTn) algorithm, that is revisited and re-engineered to bias and take over the FIRE minimization algorithm, as presently implemented in two community codes for atomistic simulations, namely Quantum ESPRESSO (PWscf) and LAMMPS. ARTn is a well established single-ended saddle-point search algorithm that allows for the exploration of potential energy surfaces. The resulting algorithm acts as a plugin, and is distributed in the form of an external library (pARTn).

Keywords: saddle point, potential energy surface, transition state, chemical reaction

Program summary

Program Title: plugin Activation Relaxation Technique nouveau (pARTn)
CPC Library link to program files: (to be added by Technical Editor)
Developer's repository link: https://gitlab.com/mammasmias/artn-plugin
Code Ocean capsule: (to be added by Technical Editor)
Licensing provisions(please choose one): Apache-2.0/GPLv3
Programming language: Modern Fortran and C++
Supplementary material:
*Journal reference of previous version:**
*Does the new version supersede the previous version?:**
*Reasons for the new version:**
*Summary of revisions:**
Nature of problem(approx. 50-250 words): original ARTn implementation was difficult to interface, port and maintain.
Solution method(approx. 50-250 words): full refactoring and re-engineering into a library, which utilizes and biases a minimization algorithm already implemented in the engine, to make it behave as ARTn. The biasing is done by reverse-engineering specific external conditions that drive the displacement of atoms.
Additional comments including restrictions and unusual features (approx. 50-250 words):

1. Introduction

Commonly used energy/force (E/F) engines in the field of atomistic simulations generally implement some integration algorithm (integrator) for solving the equations of motion (such as Verlet [1]), or for constraining the movement toward energy minimization (such as FIRE [2] and BFGS [3]). These algorithms are implemented as a series of steps enclosed in a main loop, which continues as long as an exit criterion is not met, *i.e.*, Algorithm 1. Each step of the main loop first computes the instantaneous properties of the current configuration, such as velocity or force, which are then used to generate the atomic displacement. If these instantaneous properties are modified, the resulting atomic displacement will be different. This is typically exploited for the application of external fields and constraints to the system of simulation. The same idea can however be used with other intentions: by appropriate modification of the instantaneous properties, a specifically desired atomic displacement can be imposed on the system.

In this line of thought, any algorithm present in a software could be biased by controlled modifications of the instantaneous properties, to the extent where the original purpose of the algorithm gets effectively overwritten by a different purpose, and thus the original algorithm gets taken over by another algorithm. The modification of instantaneous properties can be achieved through the application of external conditions on the system, and since the functions implementing them are gener-

ally not invasive to the native E/F engine, biasing and overwriting an algorithm in such a way is independent of the specific details of the engine, such as parallelization strategies, or the handling of charge densities. As a result, an implementation of an algorithm following this paradigm is easier to port, as well as maintain and align with respect to upgrades in E/F engine versions. A similar idea has already been implemented by PLUMED [4] for example, to steer metadynamics.

ARTn (Activation-Relaxation Technique nouveau [5, 6, 7, 8, 9]) is an open-ended saddle point search algorithm used in many different fields, ranging from materials science to molecular and biological science. Previous ARTn implementations interacted with E/F engines – such as BigDFT [10], VASP [11], Quantum ESPRESSO [9] and LAMMPS [12] – via subroutines within the main ARTn program. This implementation route was making it very difficult to handle the specifics of each E/F engine within a standardised interface, which means that separate ARTn versions had to be developed for each of the aforementioned E/F engines.

In the present work, we apply the biasing paradigm to the FIRE algorithm [2, 13], to effectively re-purpose it and make it behave as the ARTn algorithm.

To this end, the ARTn algorithm is rewritten as a library, that can be used as a plugin for different E/F engines (pARTn).

The present paper is structured as follows. Sections 2 to 5 describe the biasing/re-purposing concept, the mathematical relations, and their application to FIRE and ARTn. Section 6 describes the implementation details of the pARTn library. Section 7 is devoted to APIs/interface routines and their description for the particular case of biasing and taking over the FIRE routines in Quantum ESPRESSO [14] and LAMMPS [15]. Section 8 provides a brief package description and a quick installation guide. Finally, sections 9 and 10 describe the input and output parameters and show two working examples, respectively.

Algorithm 1: Generic integrator pseudo-algorithm. The comment in the line 3 indicates the point of entry of the re-purposing paradigm.

```
1 while continue do
2     calculate {q(i)}
3     apply external conditions                 /* bias */
4     compute ΔR( {q(i)} ), Eq. (2)             /* apply F */
5     update R(i + 1) = R(i) + ΔR, Eq. (1)
6     if exit criterion then
7         exit
8     end
9 end
```

2. The concept

Integration algorithms are generally written in terms of the positions $\mathbf{R}(i)$, and displacements $\Delta\mathbf{R}$. A generic pseudo-algorithm is shown in Algorithm 1. The atomic positions $\mathbf{R}(i)$ at step i are updated by the application of $\Delta\mathbf{R}$, as Eq. (1).

$$\mathbf{R}(i+1) = \mathbf{R}(i) + \Delta\mathbf{R} \tag{1}$$

Depending on the specific algorithm, the term $\Delta\mathbf{R}$ is some function of the set of instantaneous properties $\{\mathbf{q}(i)\} = \{\mathbf{q}_1(i),\ \mathbf{q}_2(i),\ \dots,\ \mathbf{q}_n(i)\}$, *e.g.* the force $\mathbf{F}(i)$, velocity $\mathbf{v}(i)$, or possibly others (charge, polarization, etc), and the timestep Δt:

$$\Delta\mathbf{R} = \Delta\mathbf{R}(\ \{\mathbf{q}(i)\}\ ) = \Delta\mathbf{R}(\ \mathbf{F}(i),\ \mathbf{v}(i),\ \dots,\ \Delta t) \tag{2}$$

A single iteration of the main integration loop consists of two actions: first the evaluation of the properties $\{\mathbf{q}(i)\}$, at line 2 of Algorithm 1, and second the subsequent update of the atomic positions $\mathbf{R}(i)$ to $\mathbf{R}(i+1)$, at line 5 of Algorithm 1, via the $\Delta\mathbf{R}$ obtained by the integrator (prescribed by Eq. (2)). The form of Eq. (2) is specific to the integrator algorithm used, and can be seen as application of a function F, which returns a displacement $\Delta\mathbf{R}$, from a set of given instantaneous properties $\{\mathbf{q}(i)\}$.

$$F:\ \{\mathbf{q}(i)\} \rightarrow \Delta\mathbf{R} \tag{3}$$

In order to bias an algorithm and take it over with another algorithm, the re-purposing scheme needs at least two components. Firstly, its own algorithm which prescribes a displacement $\Delta\mathbf{R}_p$, and secondly, a way to constrain and take over the host algorithm to perform the prescribed displacement $\Delta\mathbf{R}_p$ instead of $\Delta\mathbf{R}$, such that $\mathbf{R}(i+1) = \mathbf{R}(i) + \Delta\mathbf{R}_p$.

The re-purposing scheme only enters the main loop of the host algorithm once per iteration step (Algorithm 1 line 3), so it needs to be written such that each time it is called, it only prescribes one displacement $\Delta\mathbf{R}_p$, which is the displacement following its own internal algorithm.

The imposition of a prescribed displacement $\Delta\mathbf{R}_p$ on the host algorithm is achieved by modifying the properties $\{\mathbf{q}(i)\} \rightarrow \{\mathbf{q}_{mod}(i)\}$, such that the calculation of $\Delta\mathbf{R}(\ \{\mathbf{q}_{mod}(i)\}\ )$ in the host algorithm returns $\Delta\mathbf{R}_p$. In other words, the properties $\{\mathbf{q}_{mod}(i)\}$ need to be such that the application of F (Algorithm 1 line 4) returns the prescribed displacement, $F(\ \{\mathbf{q}_{mod}(i)\}\ ) = \Delta\mathbf{R}_p$. In order to obtain the proper $\{\mathbf{q}_{mod}(i)\}$, we define a function G, to be called before F, as:

$$G:\ \Delta\mathbf{R}_p \rightarrow \{\mathbf{q}_{mod}(i)\}, \tag{4}$$

which returns the set of modified properties $\{\mathbf{q}_{mod}(i)\}$, given an input $\Delta\mathbf{R}_p$, such that the subsequent $F(\ \{\mathbf{q}_{mod}(i)\}\ ) = \Delta\mathbf{R}_p$. Function G can be seen as an inverse of F, $G \sim F^{-1}$.

The function G is applied at the end of the re-purposing scheme to convert a displacement $\Delta\mathbf{R}_p$ prescribed by the internal algorithm into a set of instantaneous properties $\{\mathbf{q}_{mod}(i)\}$, such that the move performed by the host algorithm (application of F) corresponds to $\Delta\mathbf{R}(\ \{\mathbf{q}_{mod}(i)\}\ ) = \Delta\mathbf{R}_p$. See also Figure 1 for a schematic representation. The instantaneous properties $\{\mathbf{q}(i)\}$ can be modified in different ways:

a) trivially, no modification: $\{\mathbf{q}_{mod}(i)\} = \{\mathbf{q}(i)\}$, the resulting displacement will be done according to the host integrator algorithm's own logic;

b) complete overwrite: $\{\mathbf{q}_{mod}(i)\} = \{\mathbf{q}_u\}$, where $\{\mathbf{q}_u\}$ are such that $F(\ \{\mathbf{q}_u\}\ ) = \Delta\mathbf{R}_p$ is a displacement prescribed by the re-purposing function;

c) partial bias: $\{\mathbf{q}_{mod}(i)\} = \{\mathbf{q}_{mod}(\ \{\mathbf{q}(i)\}\ )\}$, the modification of the properties, and thus displacement, depends on the current "state" of the properties $\{\mathbf{q}(i)\}$.

The "bias and re-purpose" concept described above achieves its goal by only modifying the instantaneous properties of the system. As such, it is robust with respect to changes in the implementation of the host algorithm. By carefully controlling a series of such manoeuvres, the effect of the host algorithm/integrator can be completely overwritten by another algorithm, without requiring extended knowledge on the specific E/F engine implementation details.

3. Biasing the FIRE algorithm

The FIRE (Fast Inertial Relaxation Engine) algorithm [2, 13] is an efficient relaxation algorithm (minimization of energy), which is implemented in most of the E/F engines. It uses the forces and velocities computed from a molecular dynamics integration step, to constrain the update of the atomic positions and to steer the dynamics of the structure towards a minimum. The molecular dynamics integration can follow different schemes and in the case of FIRE, the semi-implicit Euler integration scheme has been shown to be one of the most robust [13]. Hence, we chose to utilize the FIRE algorithm employing that integration scheme, and in the following we discuss how the corresponding implementation is biased and re-purposed.

The atomic positions in the FIRE scheme are updated in each step with $\Delta\mathbf{R} = \mathbf{v}_{\text{eff}}(i)\Delta t$:

$$\mathbf{R}(i+1) = \mathbf{R}(i) + \mathbf{v}_{\text{eff}}(i)\Delta t, \tag{5}$$

where the effective velocities $\mathbf{v}_{\text{eff}}(i)$ are given by the modified instantaneous velocities $\widetilde{\mathbf{v}}(i)$, and instantaneous forces $\mathbf{F}(i)$, as:

$$\mathbf{v}_{\text{eff}}(i+1) = \widetilde{\mathbf{v}}(i) + \frac{\mathbf{F}(i)}{m}\Delta t. \tag{6}$$

and the $\widetilde{\mathbf{v}}(i)$ are computed in a mixing scheme of instantaneous velocities $\mathbf{v}(i)$ and forces $\mathbf{F}(i)$, such that:

$$\widetilde{\mathbf{v}}(i) = (1-\alpha)\mathbf{v}(i) + \alpha\mathbf{F}(i)\frac{||\mathbf{v}(i)||}{||\mathbf{F}(i)||}, \tag{7}$$

where α is a mixing factor. The specificity of FIRE is the use of a dot product between the forces and velocities $P = \mathbf{F}\cdot\mathbf{v}$, which determines the behaviour of the timestep Δt, and the mixing factor α. If $P > 0$ for a specified number of sequential steps, then Δt is increased, else Δt is decreased and $\mathbf{v}$ are set to zero. Conversely, if $P > 0$, then α is decreased by multiplying it with a factor, else α is reset to its original value α_0.

As it can be observed in Eq. (5), the effective $\Delta\mathbf{R}$ of the FIRE scheme is given by $\Delta\mathbf{R} = \mathbf{v}_{\text{eff}}(i)\Delta t$, which is computed directly from the instantaneous force $\mathbf{F}(i)$ (in Eq. (6) and (7)), instantaneous velocity $\mathbf{v}(i)$ (in Eq. (7)), and the mixing factor α (in Eq. (7)). Additionally, the timestep Δt is modified by FIRE itself. In the spirit of the function F from Eq. (3), the FIRE scheme can be written as

$$F_{FIRE} : \{\mathbf{q}_{FIRE}(t)\} \to \Delta\mathbf{R} \tag{8}$$

where $\{\mathbf{q}_{FIRE}(i)\} = \{\mathbf{F}(i), \mathbf{v}(i), \alpha, \Delta t\}$.

Thus, biasing the FIRE scheme is done by accessing and modifying these four instantaneous properties through a call to an external function, before inputting them to FIRE. This can be seen as the application of function F_{FIRE} with the properties $\{\mathbf{q}_{mod}(i)\}$ given from the re-purposing function (G).

If we set the function G such that the velocities $\mathbf{v}(i) = 0$ and the mixing factor $\alpha = 0$, the mixing scheme in Eq. (7) vanishes, and the function F_{FIRE} depends only on the force $\mathbf{F}(i)$, and timestep Δt.

$$F_{FIRE}(\ \mathbf{F}(i), \Delta t\ ) = \Delta\mathbf{R} = \frac{\mathbf{F}(i)}{m}\Delta t\Delta t \tag{9}$$

From the expression of Eq. (9) we can construct the function G, as follows. Given a prescribed displacement $\Delta\mathbf{R}_p$, the modified instantaneous properties are set by:

$$G(\Delta\mathbf{R}_p) = \{\mathbf{q}_{mod}(i)\} = \begin{cases} \mathbf{F}_{mod}(i) = \Delta\mathbf{R}_p m/\Delta t^2 \\ \mathbf{v}_{mod}(i) = 0 \\ \Delta t_{mod} = \Delta t \\ \alpha_{mod} = 0 \end{cases} \tag{10}$$

The function G from Eq. (10) is executed in the function applying external conditions on the system, which modifies the instantaneous properties. The displacement $\Delta\mathbf{R}$ computed by FIRE afterwards becomes equal to the prescribed displacement $\Delta\mathbf{R}_p$,

$$F_{FIRE}(\ \{\mathbf{q}_{mod}(i)\}\ ) = \mathbf{v}_{eff}(i)\Delta t = \frac{\mathbf{F}_{mod}(i)}{m}\Delta t\Delta t = \Delta\mathbf{R}_p \tag{11}$$

and the atomic positions in Eq. (1) are updated as desired, $\mathbf{R}(i+1) = \mathbf{R}(i) + \Delta\mathbf{R}_p$.

In this way, the FIRE algorithm is successfully biased and re-purposed by modifying the instantaneous properties, which are all internal to the main integration algorithm itself, and can thus be accessed and modified by an external function called right after their computation.

4. The ARTn algorithm

The ARTn algorithm itself is organised into different stages. It specifies a series of atomic displacements, or pushes, that aim to bring the structure from a local minimum to a connected saddle point of the potential energy surface (PES). Each displacement of the atoms is followed by a (partial) minimization, constrained into the hyperplane perpendicular to the displacement of the atoms. ARTn can be described as a succession of macro steps, each containing three internal actions:

1. choose the push direction;

2. push the system in that direction;

3. relax in the hyperplane perpendicular to the push direction.

Depending on which point of the PES the system is at, the push direction can be either: *(i)* the direction of the eigenvector corresponding to the lowest eigenvalue of the Hessian matrix or *(ii)* a random vector chosen at the beginning of the exploration. The exploration generally starts with the latter and the switch to the eigenvector direction happens once the lowest eigenvalue of the Hessian is negative, or lower than a prescribed threshold.

Each push is followed by relaxations in the perpendicular hyperplane. In the basin, this step prevents collisions after successive pushes, whereas outside it ensures convergence to a saddle point.

When the structure has converged to a saddle point, which is characterised by zero forces and a negative lowest eigenvalue, it is pushed into the +/- directions of the corresponding eigenvector and allowed to relax without constraints. In this way, ARTn finds a saddle point and the two minima connecting it. The lowest eigenvalue, and the corresponding eigenvector of the Hessian matrix, are computed by the Lanczos algorithm [16]. More details on ARTn can be found in Refs. [5, 9, 17].

4.1. Lanczos applied to the Hessian matrix

The Lanczos diagonalization algorithm [16] is an iterative algorithm that finds the extremum eigenvalues and eigenvectors of some matrix A, by only knowing the matrix-vector products $A\,|x_i\rangle$, where $|x_i\rangle$ are vectors generated by the Lanczos algorithm. More complete details on the algorithm can be found in the literature, see for example Appendix B in Ref. [17], or Section IIA in Ref. [18]. Most importantly, each step i of the Lanczos algorithm generates the next Lanczos vector $|x_{i+1}\rangle$, which is computed from the previous Lanczos vectors $\{|x_i\rangle\}$, such that:

$$|x_{i+1}\rangle = A\,|x_i\rangle - \alpha_i\,|x_i\rangle - \beta_{i-1}\,|x_{i-1}\rangle - \sum_{j=0}^{i} \langle x_{i+1}|x_j\rangle\,|x_j\rangle \quad (12)$$

The first term in Eq. (12) is the matrix-vector product $A\,|x_i\rangle$, while all the other terms are computed from the previous Lanczos vectors $\{|x\rangle_i\}$, as well as the coefficients α_i and β_i, which are computed at each step of the iteration and stored in a special tridiagonal matrix. The previous Lanczos vectors, and the tridiagonal matrix of α and β coefficients, are internal to the Lanczos procedure.

When the Lanczos algorithm is applied to the diagonalization of the Hessian matrix (H), the products $A\,|x_i\rangle$ can be thought of as $H(\mathbf{R}_0 + \Delta\mathbf{R}_i) = -\mathbf{F}_i$, which can be computed by the E/F engine, without knowing any elements of the H matrix. Before entering the first step of the Lanczos procedure (Lanczos step $i = 0$), the reference force $\mathbf{F}_0$ is calculated for the starting positions $\mathbf{R}_0$ and kept in memory. The Lanczos vectors generated by Eq. (12) are then made to correspond to displacements $|x_{i+1}\rangle = \Delta\mathbf{R}_{i+1}$, and the matrix-vector products are then $A\,|x_{i+1}\rangle = H\Delta\mathbf{R}_{i+1} = -\Delta\mathbf{F}_{i+1}$. Because every $\Delta\mathbf{R}_i$ uses the starting positions $\mathbf{R}_0$ as the origin, the displacement vector which we specify is modified by subtracting the current displacement $\Delta\mathbf{R}'_{i+1} = \Delta\mathbf{R}_{i+1} - \Delta\mathbf{R}_i$, thereby ensuring the structure moves from $(\mathbf{R}_0 + \Delta\mathbf{R}_i)$ to $(\mathbf{R}_0 + \Delta\mathbf{R}_{i+1})$ in a single step. After the Lanczos algorithm converges in n steps, the structure needs to return to the starting position $\mathbf{R}_0$, which is achieved by setting the displacement $\Delta\mathbf{R}_{n+1} = -\Delta\mathbf{R}_n$.

5. Repurposing FIRE to deliver ARTn

The taking over of the FIRE minimization algorithm by the ARTn algorithm using the concepts described in Sec. 3 and Sec. 4 is schematised in Figure 1. The contact point between the two algorithms is in the call to the function applying the "external conditions". This function performs two actions. First, it determines the stage of the ARTn algorithm, and computes the displacement $\Delta\mathbf{R}_p$ according to ARTn. Second, the displacement $\Delta\mathbf{R}_p$ is passed to the function G, which converts it into a set of modified instantaneous properties $\{\mathbf{q}_{mod}(i)\}$. After the call to the "external conditions" function, the properties $\{\mathbf{q}(i)\} = \{\mathbf{q}_{mod}(i)\}$ are such that the subsequent displacement performed by the host algorithm FIRE is exactly as the one prescribed by the ARTn algorithm.

Depending on the current stage of the ARTn algorithm, there are three types of displacements that can be done, which correspond to the three ways that instantaneous properties $\{\mathbf{q}(i)\}$ might be modified (see the list in Sec. 2). No modification to $\{\mathbf{q}(i)\}$ from point a) is done when allowing the system to relax from the identified saddle point. A complete overwrite $\{\mathbf{q}(i)\} = \{\mathbf{q}_u(i)\}$ from point b) is done when the system needs to be displaced with a specific known $\Delta\mathbf{R}_p$, which is the case when pushing with a given vector, and during the Lanczos iterations. Partial biasing of $\{\mathbf{q}(i)\}$ from point c) is done when the system undergoes the perpendicular relaxation, since, in this case, $\{\mathbf{q}(i)\}$ are modified by removing only the components parallel to the push vector.

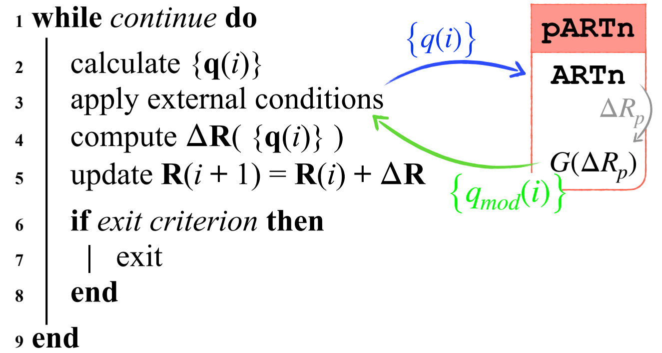

Figure 1: Schematic representation of the interaction between the pARTn library and the host algorithm. The library receives the calculated properties $\{q(i)\}$, and returns modified properties $\{q_{mod}(i)\}$ that depend on the current stage/step of the ARTn saddle point search such that the computed $\Delta\mathbf{R}$ is equal to the wanted $\Delta\mathbf{R}_p$.

5.1. Achieving specific displacements for ARTn

Corresponding to the three types of displacements in ARTn, we define three biasing functions, G_{spec}, G_{perp}, and G_{rlx}. They are as follows.

For the actions of pushing in a specific direction, and during the Lanczos iterations, the specific $\Delta\mathbf{R}_p$ are prescribed by ARTn and Lanczos algorithms, respectively. We define the biasing function $G_{spec}(\ \Delta\mathbf{R}_p\ )$, which returns the set $\{\mathbf{q}_{mod}(i)\}$ as:

$$G_{spec}(\Delta\mathbf{R}_p) = \begin{cases} \mathbf{F}_{mod}(i) = \Delta\mathbf{R}_p m/\Delta t^2 \\ \mathbf{v}_{mod}(i) = 0 \\ \Delta t_{mod} = \Delta t \\ \alpha_{mod} = 0 \end{cases} \tag{13}$$

For the action of relaxation perpendicular to the push direction, the specific $\Delta\mathbf{R}$ is a function of the instantaneous properties along that specific direction. We define G_{perp} which removes the components of the force and velocity, and leaves other properties unmodified.

$$G_{perp} = \begin{cases} \mathbf{F}_{mod}(i) = \mathbf{F}_\perp(i) \\ \mathbf{v}_{mod}(i) = \mathbf{v}_\perp(i) \\ \alpha_{mod} = \alpha \\ \Delta t_{mod} = \Delta t \end{cases} \tag{14}$$

At each first step of the perpendicular relaxation, we set $\mathbf{v}(i) = 0$ and reset the values $\alpha = \alpha_0$, and $\Delta t = \Delta t_0$. For all further steps of the perpendicular relaxation, G_{perp} is applied.

For the action of normal relaxation, the function G_{rlx} has no effect on any $\{\mathbf{q}(i)\}$, except for the first step of the relaxation, where we reset the values $\alpha = \alpha_0$, and $\Delta t = \Delta t_0$.

5.2. Algorithm control with flags and counters

The pARTn algorithm is thus composed of three biasing functions G, as given in Sec. 5.1. In order to control exactly which part of the ARTn algorithm is executed at each call to the plugin function, and which of the G functions is used to bias FIRE, we introduce counters and flags, which specify the current stage of the ARTn algorithm. For a more precise control, each stage specified by a logical flag also has one or more associated counters.

6. Implementation details

The plugin pARTn consists of three main subroutines, `artn()`, `move_mode()`, and `clean_artn()` (see the three orange blocks in Figure 2). The data received from the host algorithm enter into `artn()`, which contains the ARTn algorithm, and produce a displacement vector $\Delta\mathbf{R}_p$, in the variable `displ_vec`. This displacement vector enters in the `move_mode()` subroutine, where the biasing functions G are implemented, and data that get communicated back to the host algorithm are generated. Finally, the `clean_artn()` subroutine is there to cleanly stop the calculation by the engine.

In addition to the three main subroutines, the pARTn library also consists of two modules. The `artn_params` module, which contains the global variables used by ARTn, such as flags and counters, to keep track of the step/stage in the saddle point search, and the `units` module which contains the units employed by the different engines.

The rest of this section describes the three main subroutines from Figure 2 in greater detail.

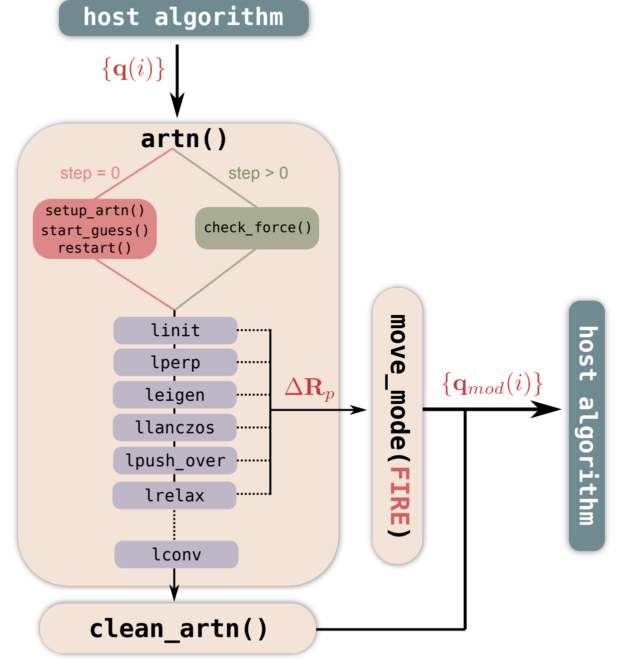

Figure 2: The organization of the three main subroutines of the pARTn library. The `artn()` subroutine receives the instantaneous properties from the host, checks convergence, determines the current stage, generates and communicates the displacement $\Delta\mathbf{R}_p$ to the `move_mode()` subroutine. The latter converts it into a set of appropriately modified properties, and passes it back to the host. The `clean_artn()` subroutine is responsible for stopping the calculation and resetting the counters/flags.

6.1. The `artn()` routine

The ARTn algorithm is contained within this subroutine, reorganized to fit the "bias and re-purpose" paradigm. This means that the subroutine is designed to be called after each calculation of the instantaneous properties, made by the host algorithm/engine, and to prescribe the subsequent ARTn step. The stage of ARTn is thus incremented each time the routine `artn()` is entered.

The core of the subroutine is divided into computational blocks; each of them corresponding to a different stage of the ARTn algorithm, and associated to a logical flag marked as `lblock`, as summarised in Figure 2. Each block can perform as a maximum number of iterations, `nblock` as controlled by an iterator `iblock`.

The saddle point search is decomposed into four blocks: `linit`, `lperp`, `leigen`, and `llanczos`. The `linit` and `leigen` blocks are responsible for displacing the configuration following either the initial push vector or the eigenvector, respectively. They both produce a specific $\Delta\mathbf{R}_p$ displacement vector in accordance to the ARTn algorithm, such that the subsequent configuration is precisely $\mathbf{R}(i+1) = \mathbf{R}(i) + \Delta\mathbf{R}_p$. The `lperp` block controls the relaxation in the hyperplane perpendicular to the push direction, this block does not generate any displacement vector, `displ_vec`, which is left to the FIRE relaxation scheme, but, rather, reduces the minimization space in $\{\mathbf{q}(i)\}$ by removing one dimension, see Eq. (14). The `llanczos` block controls the computation of the lowest eigenvalue and eigenvector with the Lanczos algorithm. It produces a specific

displacement $\Delta\mathbf{R}_p$ according to the Lanczos scheme (see Section 4.1).

Upon first entering the `artn()` routine (`step = 0`, Figure 2 top left), the input parameters are read (`setup_artn()`), and the initial displacement either read from a file, or generated on the fly (`start_guess()`). If a restart was requested, the restart file is read and the calculation parameters updated accordingly (`restart()`). The appropriate block flags are switched on/off, according to the updated parameters.

In the subsequent steps (`step > 0`, Figure 2 top right), the `artn()` subroutine first checks whether the current stage has reached convergence (`check_force()`), or if the maximum number of iterations of the current block has been reached. Depending on the outcome, the block flags `linit`, `lperp`, `leigen`, and `llanczos` are switched on/off according to what should happen next in the ARTn algorithm. When the total force is lower than the target threshold (*i.e.* the saddle point is reached), the saddle configuration is saved. Subsequently, `lpush_over` is set to `.true.` and the corresponding block is executed, displacing the atoms beyond the saddle point. Following this displacement, a simple relaxation (`lrelax`) is performed to reach an adjacent minimum. The `push_over` and `relax` blocks are repeated twice, once for each +/- sign of the push vector to obtain two minima connected to the saddle point.

6.2. *The `move_mode()` routine*

The routine `move_mode()` is responsible for transforming the displacement vector $\Delta\mathbf{R}_p$ in variable `displ_vec` that is generated in the `artn()` routine, into appropriately modified data, $\{\mathbf{q}\} \rightarrow \{\mathbf{q}_{mod}\}$, to be sent back to the host algorithm. Currently, the routine is written specifically for the case of biasing and repurposing the FIRE minimization algorithm. Depending on the stage of the ARTn algorithm, the data $\{\mathbf{q}_{mod}\}$ that are sent from `move_mode()` to the host algorithm correspond to one of the actions described in Sec. 5.1.

The choice of which action must be performed, or which of the biasing functions G must be applied (G_{spec}, G_{perp}, or G_{rlx}), is made based on the values of the block flags decided in the `artn()` routine. The function G_{spec} (Eq. (13)) is applied when either of the flags `linit`, `leigen`, `llanczos`, or `lpush_over` are turned on. The function G_{perp} (Eq. (14)) is applied when the flag `lperp` is turned on. And the function G_{rlx}, which has no action, is applied when the flag `lrelax` is turned on. The resulting set of modified properties $\{\mathbf{q}_{mod}\}$ is sent back to the host algorithm.

If an alternative host algorithm to FIRE is to be overwritten with pARTn, only the `move_mode` routine needs to be edited, in accordance with the new host algorithm.

6.3. *The `clean_artn()` routine*

This procedure is called when the algorithm has converged (`lconv`) or has been interrupted for any reason. It ensures that all variables/parameters of ARTn are reset, and ready to start a new saddle point search. All flags except `linit` are turned off, and the iterators (`iblock`) are set to zero.

When performing a series of ARTn searches, the initial atomic configuration is loaded in the actual engine position array at the start of each new ARTn research by default.

7. API and engine-specific interface

Due to the fact that E/F engines can be written in different programming languages, and because the specifics of the implementation of the host algorithm can depend on the E/F engine, an interface specific to the engine is needed.

In general, the `artn()` subroutine requires the energy/force, as well as atomic positions from the engine, and it needs to be able to modify these parameters. It also requires the total number of atoms, the order of atomic indices – which can vary among E/F engines, and the information on positional constraints specified by the user. The `move_mode()` subroutine needs to be able to modify the parameters of the host algorithm, which in the case of FIRE means having the knowledge of the initial mixing parameter (α_0) and of the initial time step (Δt_0). It also needs to be able to modify α and Δt on the fly.

Apart from these considerations, it should be noted that engines employ different units. To this end, a `units` module and a variable `engine_units` have been defined, which ensure that the units are converted to the internal units of the plugin upon input, and converted back to the units of the engine upon output. Therefore, to build an interface for the E/F engine of choice, one needs to add a case for the specific engine and define the unit conversions in this module.

This section explains how to construct an interface for two engines: (i) the Quantum ESPRESSO package for electronic structure calculations, which is based on Density Functional Theory (DFT) employing a plane wave basis set and is written in modern FORTRAN, and (ii) the LAMMPS package for Molecular Dynamics simulations using empirical interatomic potentials, which is written in C++. Integration into any other engine woud follow similar steps.

7.1. *Quantum ESPRESSO interface*

QE [19] is written in Modern FORTRAN and consists of different packages. The main package is `PWscf` (executable called `pw.x`) which computes the total energy and force of a given configuration using a Density Functional Theory (DFT) based self-consistent field (scf) approach. Since several years `PWscf` includes an empty subroutine (`plugin_ext_forces`), which is called after the energy/force calculation, that enables modifications of the instantaneous properties.

In order to utilize and re-purpose the integrator algorithm, the interface subroutine of pARTn (`artn_QE`) is placed within this empty QE subroutine (section 8.1), where it modifies the calculated properties of the current configuration in accordance with the requirements of the ARTn algorithm. The general structure of the QE interface subroutine is summarised in Algorithm 3.

QE implements several parallelisation and optimisation strategies. Since the majority of the computational time is spent for the calculation of the scf ground-state charge density, the displacement of atoms (update of atomic positions) is performed only by a single core. Similarly the pARTn routines are

Algorithm 3: Algorithm of pARTn interface with Quantum ESPRESSO

```
1  SUBROUTINE artn_qe
   Input: F, v, R, force_threshold, energy, fire_dt, fire_α,
          alat, lattice, nat, type, step, if_pos, atm,
          tmp_dir_qe, prefix_qe
   Output: F, v, R, epsf_qe, lconv
2  !> Note: Convert QE Parameters
3  box ← box* alat
4  R ← R* alat
5  order = [1, 2, ..., N_at]
6  !> The artn subroutine outputs ΔR_p
7  CALL artn( F, v, R, box, type, order, ΔR_p, lconv)
8  !> Adjust QE threshold to ARTn one
9  if (epsf_qe ≠ forc_thr) then
10 |  epsf_qe ← forc_thr
11 end
12 !> Note: Unconvert the position
13 R ← R/ alat
14 !> Note: Read FIRE Parameters
15 READ(FIREfile,*) E_tot, nsteppos, dt_curr, α
16 !> The move_mode subroutine outputs G(ΔR_p)
17 CALL move_mode( F, v, R, E_tot, nsteppos, dt_curr, α ,
     ΔR_p )
18 if ( lconv ) then
19 |  CALL clean_artn()
20 end
21 !> Note: Write FIRE parameters
22 WRITE(FIREfile,*) E_tot, nsteppos, dt_curr, α
```

meant to be called by a single mpi instance, *i.e.*, pARTn routines do not contain any mpi/openmp instruction. The internal units of QE are atomic units (Ry, bohr, a.u.t.). However, the atomic positions and lattice parameters are internally stored in atomic units scaled by a lattice parameter (`alat`). Therefore the first step of the `artn_QE()` interface subroutine is to convert the positions and lattice parameters back to atomic units, and define the order of atomic indexes that follows the order in the arrays, see Algorithm 3, lines 3, 4, and 5. After this preprocessing, all parameters are given to the `artn()` subroutine (Algorithm 3 line 7) which generates the displacement $\Delta\mathbf{R}_p$ according to ARTn algorithm. The displacement is then used as input for the `move_mode()` subroutine (Algorithm 3 line 17) which computes $G(\Delta\mathbf{R}_p) = \{\mathbf{q}_{mod}(i)\}$ for the current step, as described in Section 5.1. The force threshold of QE, `epsf_qe`, is modified to control the convergence of host algorithm. QE keeps track of the FIRE parameters by writing them to a file at each step, so that file is read before calling the `move_mode()` subroutine, and the modifications are written to it afterwards (Algorithm 3 lines 15 and 22).

Finally, the `clean_artn()` subroutine is called (Algorithm 3 line 19) when the `lconv` flag is activated by the `artn()` subroutine, which resets all the parameters of the saddle point search to their original values.

7.2. LAMMPS interface

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [15] is a package that performs molecular dynamics using empirical interatomic potentials and is written in C++. It contains an extensive library of different empirical interatomic potentials and implements a variety of minimization algorithms, including FIRE. Additionally it can easily be customized through to the activation of classes under the label `Fix`, which enable changing the rules of the simulated system, or to add some external constraints. For instance the thermodynamic ensemble of molecular dynamic can be changed by employing child classes of `Fix` class (`Class FixNVE`, `Class FixNPT`, ...). A `Fix` can be inserted in many places in the code,

Algorithm 4: Header declaration of the FixARTn class in LAMMPS

```
class FixARTn : public Fix {
    public:
    FixARTn(class LAMMPS *, int, char **);
    virtual ~FixARTn();
    int setmask();
    virtual void init();
    void min_setup(int);
    void min_post_force(int);
    void post_run();
    Protected:...
};
```

including just before the integration step. The pARTn interface for LAMMPS is a child class of `Fix`, designed to be placed in the "post force" position (`FixARTn::setmask()`). The FIRE algorithm in LAMMPS is a child class of the `min` class.

The class `FixARTn` (see Algorithm 4 for the header declaration) consists of three routines, *(i)* the routine `FixARTn::min_setup()`, which is used to initialize the FIRE parameters, and to modify the energy and force tolerance of the minimization as required by the ARTn algorithm. *(ii)* the main routine `FixARTn::min_post_force()`, in which the calls to the `artn_()` and `move_mode_()` subroutines are placed, see Algorithm 5; and *(iii)* the routine `FixARTn::post_run()`, which is called after convergence of the `min` procedure, and executes `clean_artn_()`;

The routine `FixARTn::min_post_force()` is called each time the energy and force are computed by the routine `MIN::energy_force()`. The particularity of the implementation of the FIRE algorithm in LAMMPS [13] is that the `MIN::energy_force()` can be called twice for one atomic position integration iteration, depending on the sign of the dot product of the force and the velocity (parameter $P = \mathbf{V}.\mathbf{F}$ of FIRE). Due to this setup, the ARTn algorithm cannot be incremented each time the `MIN::energy_force()` routine is called. To manage and anticipate this behaviour, the scalar

Algorithm 5: Change the force post force calculation

```
1  void FixARTn::min_post_force( int* )
   Input: vflag
2  !> Local reorder of Arrays
3  order_arrays( F, v, R, order, type );
4  Compute V.F ;
5  if (V.F ≤ 0 && step > 1 && nextblank) then
6      F ← F_prev ;
7      nextblank = false;
8      return;
9  !> Note: Prepare Arrays
10 Collect_array( F, v, R, type, order );
11 artn_(F, v, R, box, type, order, DISP, ΔR_p, lconv ) ;
12 move_mode_(F, v, R, E_tot, nsteppos, dt_curr, α , ΔR_p );
13 !> Spread the Arrays
14 Spread_Arrays( F, v, R, type, order );
15 if ( lconv ) then
16     !> Note: initial LAMMPS force threshold
           convergence
17     return;
18 !> Change FIRE parameters
19 if (DISP == 'perp' && iperp == 1) OR
20 (DISP='relx' && irelx == 1) OR
21 (DISP!='perp' && DISP!='relx') then
22     min.modify_params();
23     min.init();
24 end
25 Compute V.F;
26 if (V.F ≤ 0) then
27     nextblank = true;
28 if (step == 0) then
29     nextblank = false;
30 F_prev ← F;
31 return
```

product P is explicitly calculated, the modified forces vector $\mathbf{F}$ returned by ARTn is saved from one iteration/call to another in $\mathbf{F}_{\mathbf{prev}}$, and re-loaded into $\mathbf{F}$ when P is negative. So at the beginning of the routine `FixARTn::min_post_force()`, the parameter P is explicitly computed (Algorithm 5 line 4), and if P is negative, $\mathbf{F} = \mathbf{F}_{\mathbf{prev}}$ and returned to the calling function, otherwise it enters in the ARTn procedure. At the end of the routine `FixARTn::min_post_force()` the returned force is saved, $\mathbf{F}_{\mathbf{prev}} = \mathbf{F}$. To work with both serial and parallel applications, the algorithm starts by collecting the data distributed among processors, due to MPI paradigm, in arrays (`Collect_array()`), as well as the internal order of atoms in LAMMPS. The order of atoms in LAMMPS arrays can change due to the distribution of data over the processors in parallel execution, however the pARTn library is executed serially. The forces and velocities of all atoms, along with the order of indexes, are collected and given to `artn_()`, followed by `move_mode_()`. Afterwards, the new vectors of force and velocity are spread through all the processors (`Spread_arrays()`). If convergence is reached (`lconv`) then the interface returns to the calling function. Otherwise the FIRE parameters are updated according to the specific step of ARTn described by variable `DISP`.

7.3. Summary for the building of an interface

The two presented interface examples showcase the organisation of calls to the three main routines of pARTn, *i.e.*, `artn()`, `move_mode()`, and `clean_artn()`. In general, an interface should contain a step in which the arrays that get passed to `artn()` and `move_mode()` are organised in accordance with their requirements. Originally these arrays can be in any engine-specific format, which can be due to the distribution of data between processors in MPI paradigm, or other specific engine data management. After these arrays are used by pARTn, an additional step is needed, where they are converted back to the format required by the engine. A call to the routine `clean_artn()` at the end ensures that the saddle point search is stopped correctly, and the ARTn variables, flags, and counters are reset for a new search.

8. Package description and documentation

The pARTn library package contains the following directories and building files:

- `src/` (folder): containing the source code of the library;
- `Files_LAMMPS/` (folder): containing the files and scripts to interface ARTn with LAMMPS;
- `Files_QE/` (folder): containing the files and script to interface ARTn with Quantum ESPRESSO;
- `examples/` (folder): examples of saddle point searches using QE and LAMMPS;
- `environment_variables` (file): File defining the environmental variables;
- `Makefile` (file): Contains compilation instructions using the `make` command;
- `README.md` (file): Contain a description and compilation instructions for both interfaces.
- `TERMS_OF_USE` (file)
- Textual copy of the License

The online documentation with further details is available at the link: https://mammasmias.gitlab.io/artn-plugin/.

8.1. Quick Installation guide

8.1.1. Compile pARTn

The ARTn-plugin-v1.0.0.tar.gz package can be obtained either from the journal repository, or from the pARTn git repository[1]. To compile pARTn, specific variables must be configured

[1] https://gitlab.com/mammasmias/artn-plugin/-/releases

in the environment_variables file. These variables include the names of the compilers (F90, C, and C++ compilers defined in F90, CC, and CXX, respectively), the corresponding run command (PARA_PREFIX) if required, the paths to the E/F engine(s) (Quantum-ESPRESSO and/or LAMMPS main directory), and the BLAS library path. It is essential to use the same compiler that was used to compile the engine, due to the shared library interface between the engine and pARTn library. The main pARTn directory contains a Makefile, and all available make options can be displayed by typing the appropriate command.

```
make [help]
```

with or without the argument help.

8.1.2. *pARTn for Quantum ESPRESSO*

In the following, a quick Quantum ESPRESSO 7.0 installation guide is provided, followed by instructions on how to patch QE with pARTn. For more specific QE installation instructions, please refer to the QE documentation. Download Quantum ESPRESSO 7.0 from the QE web site (www.quantum-espresso.org). Untar and unzip. In the Quantum ESPRESSO main directory type:

```
./configure
make pw
```

In the pARTn main directory the command

```
make lib
```

will compile libpart.a, which is the library needed for the QE/pARTn interface.

```
make patch-qe
```

will patch and recompile Quantum ESPRESSO with the libpart.a dependency.

A calculation invoking pARTn can be run by using the flag -partn as argument of pw.x:

```
mpirun -np N pw.x -partn -inp input.qe
```

Note that patches have been tested ONLY for Quantum ESPRESSO 7.0, which is the first version implementing FIRE. For patching pARTn on other QE versions, please contact us.

8.1.3. *QE input specification*

For a proper execution of pARTn within QE, three variables must be specified in the QE input file, namely: in the CONTROL namelist, the calculation type must be specified as calculation = "relax", and the use of symmetries disabled nosym = .true., and in the IONS namelist, the dynamics must be specified as ion_dynamics="fire".

8.1.4. *pARTn for LAMMPS*

In the following a quick LAMMPS installation guide is described, followed by instructions on how to compile pARTn as a shared library, which is needed in order to behave as a LAMMPS plugin. For more specific LAMMPS installation instructions please refer to the LAMMPS documentation.

First a LAMMPS 23 june 2022 or a more recent version needs to be downloaded from the official LAMMPS website (www.lammps.org). Untar and unzip. Before compiling LAMMPS, the PLUGIN package needs to be activated. In the directory LAMMPS/src/, type:

```
make yes-plugin
```

then LAMMPS compiled in the preferred way, using make, or CMake, more info: https://docs.lammps.org/Build.html. For a standard X86 linux, typing

```
make mode=shared serial (or mpi)
```

from the src directory should work.

The LAMMPS/pARTn interface uses the shared library libartn.so, that is built by the command

```
make sharelib
```

from the main pARTn directory.

Note that the compiler (CC or CXX) defined in the file environment_variables should be the same as the one used to compile LAMMPS. As stated earlier, the PLUGIN package of LAMMPS needs to be activated (make yes-plugin), which is available in LAMMPS since the version stable_23Jun2022.

8.1.5. *LAMMPS input specification*

In the LAMMPS input script, the pARTn library passes through the class plugin. The fix artn can be used only after loading the dynamic library libartn.so, as for example:

```
plugin load /path/to/pARTn/libartn.so
fix fix_ID all artn
min_style fire
minimize etol ftol maxiter maxeval
```

The fix artn must also be associated with the algorithm FIRE that is defined by the min_style command.

8.1.6. *delete_atoms and order in LAMMPS*

It must be noted that the order array of atomic indexes which enter pARTn must be contiguous, or more precisely, the maximal atomic index must correspond to the size of the array of the positions. Attention when using the delete_atoms function of LAMMPS – the keyword compress yes should also be used.

9. Input and Output

9.1. *I/O Format*

Along with the regular input/output file(s) from the E/F engine, pARTn has its own human-readable input and output

files, artn.in, artn.out, and files containing the found saddle/minima configurations.

The pARTn input file artn.in is formatted as a Fortran NAMELIST, containing the input parameters of ARTn. The pARTn output file artn.out contains precise data on the progress of the current saddle point search. The files containing found saddle/minima configurations are text files written in two possible formats, xsf or xyz, specified by the input variable struc_format_out. The filenames of found configurations are printed at the end of each ARTn research in the output file artn.out. In order to not overwrite any of the configuration files during an extensive exploration, counters are used as part of the configuration filenames, with prefixes min or sad, depending on whether the configuration has been found as a minimum, or saddle. For example the filename sad0013 indicates the configuration written in the file is a saddle point number 13.

9.2. Input description

The pARTn input file 'artn.in' is used to define/modify the ARTn variables, which are presented in the Table 1. This file is read in the initialization step of the artn() routine, specifically by the subroutine setup_artn().

The parameters controlling the general behaviour of pARTn are verbose, engine_units, lrestart, struc_format_out, and converge_property. The verbosity of the output is controlled by either value verbose={0,1,2} , where 0 is the least verbose. The units of the E/F engine are specified through engine_units, for instance engine_units='qe' when QE is used. The logical flag lrestart=.true. can be set when a restart from a previous calculation is desired. There are two possibilities for setting up the convergence criteria, converge_property='maxval' or converge_property='norm', which decides whether the convergence value is compared to the maximum value or to the total norm of the property being checked for convergence (force).

Each block of the ARTn algorithm can be customised to some extent, by modification of the relevant parameters in the input file.

9.2.1. Controlling the initial push

The initial push vector can be customised with the combination of five parameters: push_mode, push_ids, add_const, push_step_size, and dist_thr. The parameter push_mode specifies the way to setup the initial push vector, and it has possible values all, list, rad, or file. In the case push_mode='all', the initial push vector is generated containing a push on all the atoms present in the system; when push_mode='list' it is generated only for a list of atoms, which needs to be provided as push_ids=id1,id2, ... , where id# are the indices of atoms where the push vector shall be nonzero.Doctor In order to define a preferential direction for the initial random push of the atoms, the parameter add_const can be used, for example the command add_const(:,id1) = 1.0, 0.5, -1.0, 30.0 will constrain the push vector on the atom id1 to be generated within a circular cone with the axis in the $(1.0, 0.5, -1.0)$ direction, and the angle 30 degrees from its apex, where the apex is the position of the atom. The push_mode='rad' is used when a group of atoms within a radius dist_thr of each atom specified in push_ids should have a nonzero push vector. When the initial push vector should be read from a file, the push_mode='file' is used, with the filename of the push vector provided in push_guess=filename.xyz specified. The norm of the initial push vector is regulated by the push_step_size parameter, except for when reading it from a file (push_mode='file') where the vector is used as-is. The format of the xyz file is explained in Appendix A.

9.2.2. Controlling the number of pushes before calling Lanczos

The evaluation of the lowest eigenvalue with the Lanczos procedure can consume a non-negligible portion of the computational time, due to many force computations, whereas it is sometimes not needed. This is typically the case in the starting basin, where the lowest eigenvalue is evidently positive. To avoid force calculations in that case, it is possible to set the integer variable ninit, which specifies the minimal number of pushes to be done with the initial push vector, without calling Lanczos. When the number of pushes exceeds ninit, the Lanczos procedure is called for the first time. During the first ninit steps, ARTn is thus effectively blind to the lowest eigenvalue and the corresponding eigenvector.

It is also possible to set ninit=0, in which case ARTn will directly enter the Lanczos scheme from the start, and evaluate the pushing direction based on the obtained eigenvalue. This is useful when the starting structure is already close to a saddle configuration, where the lowest eigenvalue is very likely negative, and the structure only needs to be refined to a saddle.

9.2.3. Controlling the Lanczos algorithm

For controlling the Lanczos diagonalisation scheme, parameters lanczos_disp, lanczos_max_size, and lanczos_eval_conv_thr are used. The lanczos_disp controls the norm of the displacement $\Delta\mathbf{R}_i$ prescribed by each Lanczos vector, to compute $\Delta\mathbf{F}_i$ in each iteration step. The lanczos_max_size prescribes the maximum number of iterations of the Lanczos scheme, which is also the maximal size of the tridiagonal matrix of α_i and β_i coefficients getting diagonalized at each iteration. The lanczos_eval_conv_thr is the convergence threshold on the eigenvalue λ_i obtained at each Lanczos iteration i. The convergence criterion is:

$$\left|\frac{\lambda_i - \lambda_{i-1}}{\lambda_{i-1}}\right| \leq \texttt{lanczos_eval_conv_thr} \qquad (15)$$

Once λ_i is converged, the Lanczos scheme exits, or alternatively when the maximal number of iterations i =lanczos_max_size is reached. For the first ARTn step where Lanczos scheme is called, the starting Lanczos vector is random. Alternatively, it can be read from a file specified by eigenvec_guess, written in the xyz format, *i.e.* Appendix A, with atomic types replaced with atomic indexes. For the subsequent ARTn steps, the eigenvector calculated in the previous ARTn step is reused as the first vector for the current Lanczos scheme.

9.2.4. *Controlling the eigenvector push*

Once the eigenvalue obtained is lower than a prescribed threshold eigval_thr, the push vector is overwritten by the corresponding eigenvector. The maximal size of each push with the eigenvector is regulated with eigen_step_size. In order to make a smooth transition from pushing with the initial push vector to pushing with the eigenvector, the parameter nsmooth can be used, for instance nsmooth=3 indicates the transition will be done in 3 steps, where during the transition the push vector is linearly interpolated between the initial push and eigenvector (see Ref [9]), the default value is however nsmooth=0.

It can happen during the saddle search, that despite all efforts, a negative eigenvalue becomes positive, without ever passing a saddle, and in a region which is far from a minimum. In this case the direction of the eigenvector cannot be trusted anymore to be pointing in the direction of a saddle. The default action is thus to abort the current search, and start from the beginning. An attempt to mitigate the behaviour in that scenario is however to re-set the initial Lanczos vector to a random vector, and attempt to recompute the true eigenvector. The number of times this resetting is allowed to happen during a single search is controlled by the variable nnewchance, which is by default nnewchance=0.

9.2.5. *Controlling the number of steps in the perpendicular relaxation*

To avoid falling back to the initial minimum and to save some unnecessary computational effort, it is recommended to control the number of perpendicular relaxation steps for each ARTn macro step. These numbers are set by the array nperp_limitation, for which the default values are $(4, 8, 12, 16, -1)$. If the structure is still in the basin – lowest eigenvalue is positive or above the corresponding threshold, then the number of perpendicular relaxation steps is given by the first number in the nperp_limitation array. Once the structure is out of the basin – lowest eigenvalue is negative or below threshold, the number of perpendicular relaxation steps is gradually increased in each ARTn macro step, as prescribed by the sequence given by the nperp_limitation array. In the case of the default values, the sequence would be: 8 perpendicular relaxations after the first eigenvector push, 12 after the second, 16 after the third. The number of entries in the array is free, thus adding more entries to the sequence is possible. The last entry of the nperp_limitation array indicates the number of relaxation steps to be done when the sequence runs out. With the value -1, the code will perform perpendicular relaxations until the perpendicular component of the force is lower in magnitude than the parallel component.

For fixing the number of perpendicular relaxation steps while the structure is still in the basin, the input variable nperp can also be used. If set, the value of nperp will be pre-pended to the nperp_limitation array.

9.2.6. *ARTn force threshold*

The forces are assumed to be a $3N$ vector in which N is the number of atoms. The convergence of forces is not tested while the eigenvalue is positive (structure is still in basin). It starts to get tested once the lowest eigenvalue becomes negative, during the perpendicular relaxation (possible convergence to the saddle), and during the regular relaxation to the adjacent minima, following the criterion prescribed by the input variable forc_thr. The configuration has converged to either a saddle point, or a minimum, when the sum of the parallel and perpendicular components of all atomic forces is lower than forc_thr.

The convergence of forces is tested by computing the absolute size of the forces vector, and comparing it to the threshold forc_thr. The absolute size of the forces vector can be computed in two ways, either as maximal absolute element of the vector, or as Cartesian 2-norm of the vector. The choice between the two is prescribed by setting the variable converge_property='maxval' or 'norm'.

The convergence criterion forc_thr can be very system-dependent, as well as E/F engine-dependent, and it should therefore always be tuned accordingly.

9.2.7. *Final push, move to next minimum*

Once a saddle point is reached, the default ARTn behaviour also involves the final push and relaxation to the two adjacent minima. This is controlled by the logical flag lpush_final, which when set to .false., will signal ARTn to stop once a saddle point has been found, and reset the atomic positions to the initial configuration. On the other hand, if lpush_final = .true., the algorithm will use the eigenvector obtained at saddle to push the configuration to the +/- directions, and then allowed to relax. The size of this final push is regulated through push_over parameter.

Once the two minima are obtained, it is possible to set the atomic positions to the adjacent local minimum, which is achieved by setting the flag lmove_nextmin=.true.. As such the subsequent ARTn exploration starts from the new minimum configuration, otherwise the atomic positions are reset to the initial configuration.

9.3. *Output summary/or Output description*

Along with the regular E/F engine output, the pARTn calculation produces another output file, named 'artn.out'. This file contains a summary of input variables in the header, and information about the current ARTn search. The output quantities are given in the units defined as engine units, specified by the engine_units variable. The quantities written in output at the most verbose level are, in respective order:

- istep: Iteration step of ARTn;
- ARTn_step: Computation block;
- Etot: Energy difference from the initial configuration;
- init/eign/perp/lanc/relx: Iterator "iblock" for each computational block;
- Ftot Fperp Fpara: The value of total force and its perpendicular and parallel components;
- eigval: The minimum eigenvalue computed by lanczos;

- `delr`: The total displacement of the configuration from the initial configuration;
- `npart`: Number of particles moved from the initial configuration;
- `evalf`: Number of force evaluations;
- `a1`: scalar product between the current and the previous push direction.

A brief report is written at the convergence to the saddle point, indicating the energy difference of the saddle with respect to the initial configuration, and the filename in which the saddle configuration is saved. A similar report is written at the convergence to minima from the saddle point. At the end of the search, a debriefing line is written, which summarises the search.

In addition to `artn.out`, the files containing found configurations are generated. These files are written in the format requested (`struc_format_out='xsf'`, or `'xyz'`), and contain information on the number of atoms, lattice vectors, total energy, atomic types, atomic positions, and the forces. The filenames produced during each ARTn search are written at the end of the output `artn.out`.

At the lowest level of verbosity `verbose = 0`, only the configuration filenames are printed in the output.

10. Examples

The repository includes several examples (in the directory `examples`), employing both currently interfaced E/F engines. The examples with QE as the E/F engine are based on the ones presented in the previous ARTn *ab initio* implementation [9, 17]. The examples with LAMMPS as a E/F engine are taken from OptBench [20] datasets, or from the former ARTn repository[2], or have been designed for the purpose of pARTn. In the following we describe two examples in detail, one for each interface: the diffusion of an Al adatom on the Al(100) surface with QE, and a search for saddle point of a Pt heptamer island on Pt(111) surface with LAMMPS. To run the examples it is first necessary to have a compiled version of the codes.

10.1. QE: Diffusion of an Al adatom on the Al(100) surface

This example is designed to demonstrate a constrained saddle search with the pARTn plugin by employing QE as the E/F engine. The instructions to compile and patch QE with pARTn can be found in sections 8.1.1 and 8.1.2.

The example can be found in the `/examples/Alad.Al100.QE/` folder. The initial structure is a relaxed configuration of an Al adatom adsorbed in a hollow site in a (5×5) supercell of a 6 layer Al(100) slab. The two bottom layers of the Al(100) are constrained to their bulk positions, whereas all other layers are allowed to relax. The calculation is performed at the PBE level of theory, employing a plane wave basis set with a 35 Ry cutoff for the wavefunctions (280 Ry for the charge-density) in combination with an ultrasoft pseudopotential for Al (file `Al.pbe-n-rrkjus_psl.0.1.UPF`[3]). The Brillouin zone is sampled only at the Gamma point. The QE input file (`artn.Al-hollow.Al100-5x5-6l.in` is modified in order to work with the `pARTn` plugin, the key modifications being to set `calculation = 'relax'`, `nosym = .true.`, and `ion_dynamics = 'fire'` in the appropriate namelists. A shell script (`run_example.sh`) is provided that launches the example automatically.

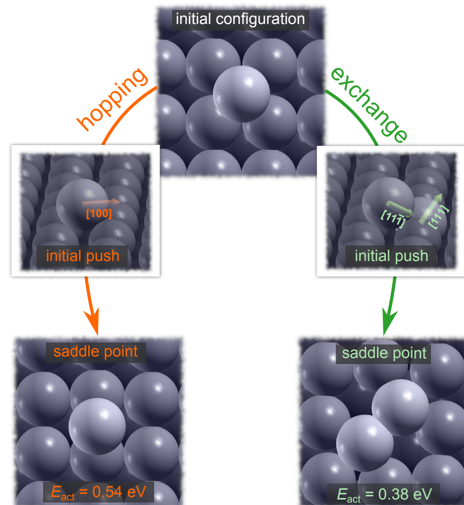

Figure 3: The initial configuration and the two initial displacements that lead to the identification of the the saddle points corresponding to the hopping and exchange mechanism for the diffusion of the Al adatom on the Al(100) surface. The topviews of the saddle point configurations are also shown, the calculated activation energies (E_{act}) are 0.54 eV for hopping and 0.38 eV for exchange.

The aim of the example is to calculate the energy barrier associated to two different mechanisms for the diffusion of an Al adatom to an adjacent hollow site. The first mechanism (hopping) involves only the displacement of the Al adatom from one hollow site to another via the bridge site. The second mechanism (exchange) involves the adatom and its nearest neighbour, where the adatom moves to the positions of its nearest neighbour and the nearest neighbour is displaced to the adjacent hollow site. This is a well known example and for more details on different mechanisms please see Refs. [21, 22].

Therefore, two constrained saddle searches (as shown in Figure 3) are performed with the `pARTn` plugin. In order to constrain the initial displacement to a specific subset of atoms we set `push_mode = 'list'` in `artn.in`. For the hopping mechanism the initial displacement is placed only on the adatom (atomic index 1) by setting `push_ids = 1` and the displacement is constrained to the [100] direction, by specifying `add_const(1) = 1.0, 0.0, 0.0, 0.0`. All other parameters retain their default values. On the other hand, for the exchange mechanism, the adatom (atomic index 1) and one if its nearest neighbours are displaced (atomic index 14), by set-

[2] https://normandmousseau.com/ART-nouveau.html

[3] `http://pseudopotentials.quantum-espresso.org/upf_files/Al.pbe-nl-rrkjus_psl.1.0.0.UPF`

ting push_ids=1,14, the displacement of the adatom is done in the $[11\bar{1}]$ direction (towards its nearest neighbour), and the nearest neighbour is displaced in the [111] direction (towards a hollow site). Therefore, the constraints are specified as add_const(:,1) = 1.0,1.0,-1.0, 0.0 for the adatom, and add_const(:,14) = 1.0, 1.0, 1.0, 0.0 for the nearest neighbour. Additionally, the eigenvalue of the characteristic eigenvector is above the default eigval_thr, therefore the latter is set to eigval_thr = -0.005 for this calculation. Following this setup, the two saddle points are reliably identified. Note however that the calculations for this example are computationally quite demanding and should be run by using the parallelization capabilities of QE, i.e., by using 16 cores (Intel-Xeon W-2295 CPU 3.00GHz) a saddle search consumes about 5 hours of wall time, mainly due to the force calculation. The results of this example are two identified saddle points, one for the exchange mechanism of Al adatom diffusion, with a barrier of 0.38 eV and one for the hopping mechanism, with a barrier of 0.54 eV (see Fig. 3). The files generated in a succesfull run are available in the reference.d folder.

10.2. LAMMPS: Pt(111) surface heptamer island

This example shows the basic use of the pARTn plugin with the LAMMPS E/F engine. It can be found in the folder /example/Pt111_lammps/. The instructions to compile pARTn as a LAMMPS plugin can be found in sections 8.1.1 and 8.1.4.

The structure is a Pt(111) surface with a heptamer island of adatoms on top, which is in an energetic minimum, see Figure 4 left. It is taken from the OptBench database [20], from the section of saddle search benchmark. The interatomic potential used is the Morse potential, with $D = 0.7102$, $\alpha = 1.6047$, and $r_0 = 2.897$.

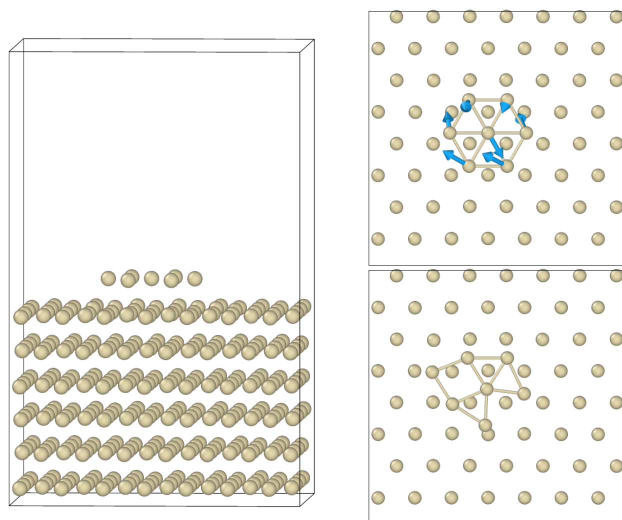

Figure 4: Left: side view of the Pt(111) heptamer structure. Top right: top view of the heptamer island and the first layer of atoms underneath, with the initial push vector marked in blue arrows. Bottom right: the structure of the saddle point found with the given push. The atomic bonds are drawn only between the heptamer island atoms for clarity.

The example is launched in a similar way as any other LAMMPS calculation using the FIRE minimization. In order to specify the pARTn calculation, the plugin must be loaded and invoked through the fix as explained in section 8.1.5. There is a maximum size for the displacement step in the FIRE implementation in LAMMPS (variable dmax), which can be modified by adding it as parameter when invoking the fix, *e.g.*: "fix fix_ID all artn dmax 3.0". All other parameters of FIRE can be edited in the same fashion.

To communicate the units of the E/F engine to pARTn, the command engine_units='lammps/metal' is specified in the artn.in input file for pARTn. Note that all parameters given in this input are now relative to the specified units. The condition for signalling a converged saddle point for this example is that the norm of the total force is below 10^{-3} eV/Å, which is specified with the commands converge_property='norm', and forc_thr=0.001. The parameters for computation of the eigenvalues and eigenvectors are as follows. The displacement size lanczos_disp=1e-4 for the Lanczos vectors, the maximal number of iterations lanczos_max_size=10, and the threshold of convergence of the eigenvalue lanczos_eval_conv_thr=1e-2, *i.e.* section 9.2.3. The threshold for the eigenvalue is eigval_thr=-0.02. In order to smooth the transition from pushing with the initial push vector, to pushing with the found eigenvector, the parameter nsmooth=2 is used, which signals the transition is done in 2 steps.

The main feature of this example is to read the initial push from a file for the 7 atoms of the cluster, and use it to find a saddle point. This is done by the input commands push_mode='file' followed by push_guess='ini_push.xyz', which specifies the filename to read from. The format of the file is explained in Appendix A. The given initial push vector of this example is shown in Figure 4 top right in blue arrows, resized by factor 3 for better visibility. This vector is used to push the structure ninit=1 times before computing the lowest eigenvalue for the first time.

With the given input parameters and the push vector, the result should be a saddle point with the structure as shown on the bottom right of Figure 4, with the energy barrier of 1.47 eV. Notice that the atom at the center of the hexagon had some displacement prescribed by the initial push vector, but in the saddle point this atom is not displaced much from its minimum position. This indicates that the initial push vector does not need to be extremely precise to find a saddle.

In the Supplementary Materials of this article, a self-contained python notebook is available (.ipynb file), which downloads LAMMPS and pARTn, compiles both, and launches the Pt(111) example within a python environment.

11. Conclusion

In the present work we describe a paradigm to bias and repurpose integrator algorithms already present in E/F engines, and overwrite them with a different algorithm, in the context of atomistic simulations. This paradigm works by modifying the instantaneous properties of the system, which are accessed and modified via the function designed to apply the external

conditions on a system. Since these functions are generally not invasive, biasing and taking over an algorithm in this way is independent of the specific details of the native E/F engine. This paradigm results in a plugin-algorithm which is easier to port, maintain, and align with respect to the upgrades in the E/F engines. As a proof of concept, we present a complete refactoring of the ARTn algorithm, into a plugin library (pARTn), in line with the hijacker "bias and re-purpose" paradigm. We show its porting and application to Quantum ESPRESSO [14] and LAMMPS [15]. The pARTn library is a double licensed Apache-2.0/GPLv3 and can be downloaded from git repository [4].

ACKNOWLEDGEMENTS

N.M.'s work is supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. This work was performed using HPC resources from CALMIP (Grant P1418). M.P. M.G., N.S., A.J., A.H., N.R., N.M., L.M.S. are active members of the multiscale and multimodel approach for materials in applied science consortium (MAMMASMIAS consortium) and acknowledge the efforts of the consortium in fostering scientific collaboration.

Appendix A. Guess input vector format

The vector is given in the standard xyz format, where the first line is the number of atoms in the list, second line is empty (comment), and the list of atoms starts at the third line. The format of the list is flexible. The first argument of each line must be the atom index. If only the atom index is given then the displacement for that atom will be random, otherwise three numbers must be given following the atomic index, which represent the coordinates of the push for that atom. An example of the push init file is given for the 7 atoms of the LAMMPS example from Sec. 10.2.

```
7

1  -7.58E-003   6.22E-002  -2.90E-002
2   5.44E-002  -6.24E-002  -2.90E-002
3   5.20E-002  -5.09E-002  -3.32E-002
4   5.92E-002  -6.22E-002  -4.53E-002
5  -2.61E-002  -1.24E-002   3.77E-002
6  -5.06E-002  -4.33E-002  -1.31E-002
7   4.71E-002  -3.91E-002   3.19E-002
```

Note that upon reading the push vector from a file, it is used as-is, without any rescaling or other modification, thus it needs to be given in units of ARTn, which is in Bohr radius.

References

[1] L. Verlet, Phys. Rev. **159**, 98 (1967).

[2] E. Bitzek, P. Koskinen, F. Gähler, M. Moseler, and P. Gumbsch, Phys. Rev. Lett. **97**, 170201 (2006).

[3] R. Fletcher, Conjugate direction methods, in *Practical Methods of Optimization* (John Wiley & Sons, Ltd, 2000) Chap. 4, pp. 80–94, https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118723203.ch4 .

[4] M. Bonomi, G. Bussi, C. Camilloni, G. A. Tribello, P. Banáš, A. Barducci, M. Bernetti, P. G. Bolhuis, S. Bottaro, D. Branduardi, R. Capelli, P. Carloni, M. Ceriotti, A. Cesari, H. Chen, W. Chen, F. Colizzi, S. De, M. De La Pierre, D. Donadio, V. Drobot, B. Ensing, A. L. Ferguson, M. Filizola, J. S. Fraser, H. Fu, P. Gasparotto, F. L. Gervasio, F. Giberti, A. Gil-Ley, T. Giorgino, G. T. Heller, G. M. Hocky, M. Iannuzzi, M. Invernizzi, K. E. Jelfs, A. Jussupow, E. Kirilin, A. Laio, V. Limongelli, K. Lindorff-Larsen, T. Löhr, F. Marinelli, L. Martin-Samos, M. Masetti, R. Meyer, A. Michaelides, C. Molteni, T. Morishita, M. Nava, C. Paissoni, E. Papaleo, M. Parrinello, J. Pfaendtner, P. Piaggi, G. Piccini, A. Pietropaolo, F. Pietrucci, S. Pipolo, D. Provasi, D. Quigley, P. Raiteri, S. Raniolo, J. Rydzewski, M. Salvalaglio, G. C. Sosso, V. Spiwok, J. Šponer, D. W. H. Swenson, P. Tiwary, O. Valsson, M. Vendruscolo, G. A. Voth, A. White, and T. P. consortium, Nature Methods **16**, 670 (2019).

[5] G. T. Barkema and N. Mousseau, Phys. Rev. Lett. **77**, 4358 (1996).

[6] R. Malek and N. Mousseau, Phys. Rev. E **62**, 7723 (2000).

[7] H. Kallel, N. Mousseau, and F. m. c. Schiettekatte, Phys. Rev. Lett. **105**, 045503 (2010).

[8] M.-C. Marinica, F. Willaime, and N. Mousseau, Phys. Rev. B **83**, 094119 (2011).

[9] A. Jay, C. Huet, N. Salles, M. Gunde, L. Martin-Samos, N. Richard, G. Landa, V. Goiffon, S. De Gironcoli, A. Hémeryck, and N. Mousseau, *Journal of Chemical Theory and Computation*, J. Chem. Theory Comput. **16**, 6726 (2020).

[10] E. Machado-Charry, L. K. Béland, D. Caliste, L. Genovese, T. Deutsch, N. Mousseau, and P. Pochet, The Journal of Chemical Physics **135**, 034102 (2011), https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/1.3609924/13319917/034102_1_online.pdf .

[11] N. Salles, N. Richard, N. Mousseau, and A. Hemeryck, The Journal of Chemical Physics **147**, 054701 (2017).

[12] M. Trochet, L. K. Béland, J.-F. m. c. Joly, P. Brommer, and N. Mousseau, Phys. Rev. B **91**, 224106 (2015).

[13] J. Guénolé, W. G. Nöhring, A. Vaid, F. Houllé, Z. Xie, A. Prakash, and E. Bitzek, Computational Materials Science **175**, 109584 (2020).

[14] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. D. Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch, Journal of Physics: Condensed Matter **21**, 395502 (2009).

[15] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, Comp. Phys. Comm. **271**, 108171 (2022).

[16] C. Lanczos, Journal of Research of the National Bureau of Standards **45**, 255 (1950).

[17] A. Jay, M. Gunde, N. Salles, M. Poberžnik, L. Martin-Samos, N. Richard, S. de Gironcoli, N. Mousseau, and A. Hémeryck, Computational Materials Science **209**, 111363 (2022).

[18] R. A. Olsen, G. J. Kroes, G. Henkelman, A. Arnaldsson, and H. Jónsson, The Journal of Chemical Physics **121**, 9776 (2004), https://doi.org/10.1063/1.1809574 .

[19] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. B. Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. D. Corso, S. de Gironcoli, P. Delugas, R. A. DiStasio, A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, J. Jia, M. Kawamura, H.-Y. Ko, A. Kokalj, E. Küçükbenli, M. Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H.-V. Nguyen, A. O. de-la Roza, L. Paulatto, S. Poncé, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu, and S. Baroni, Journal of Physics: Condensed Matter **29**, 465901 (2017).

[20] S. T. Chill, J. Stevenson, V. Ruehle, C. Shang, P. Xiao, J. D. Farrell, D. J.

[4] https://gitlab.com/mammasmias/artn-plugin

Wales, and G. Henkelman, Journal of Chemical Theory and Computation **10**, 5476 (2014), pMID: 26583230, https://doi.org/10.1021/ct5008718 .
[21] G. Henkelman and H. Jónsson, J. Chem. Phys. **111**, 7010 (1999).
[22] T. Fordell, P. Salo, and M. Alatalo, Phys. Rev. B **65**, 233408 (2002).

name	type	default	description
I/O and calculation options			
verbose	INT	0	3 levels of verbosity: $\{0, 1, 2\}$;
engine_units	CHAR	'qe'	Units used by the engine: "qe": Rydberg, bohr, a.u.time; "lammps/metal": eV, Å, ps;
struc_format_out	CHAR	'xsf'	Output structure format, xyz or xsf;
lrestart	BOOL	F	Flag for restarting a search, if set to T, the restart file (artn.restart) is read;
lpush_final	BOOL	T	When T, relax to both minima adjacent to the saddle point;
lmove_nextmin	BOOL	F	Reset the configuration to that of the final minimum when the ARTn algorithm is finished;
Controlling initial push			
push_mode	CHAR	'all'	Possible values are 'all', 'list', 'rad', or 'file';
push_ids	INT(:)	$(\mathrm{id}_1, .., \mathrm{id}_N)$	List of atom indices with nonzero components in the initial push vector;
add_const	REAL(4,:)	0.0	Constraint on the initial push on each atom: 3-component vector, and 1 solid angle in degrees;
dist_thr	REAL	0.0	Generate push on all atoms within the radius from an atom in push_ids, used in combination with push_mode = 'rad';
push_step_size	REAL	0.3	Maximum size of a component in the initial displacement ($\Delta\mathbf{R}_{\mathrm{init}}$);
push_guess	CHAR	" "	Filename to read the initial push vector, used in combination with push_mode='file';
ninit	INT	3	Number of initial displacements without calling Lanczos;
Controlling the lanczos algorithm			
lanczos_max_size	INT	16	Maximum number of Lanczos iterations;
lanczos_disp	REAL	10^{-2}	Scaling factor for displacement during the Lanczos algorithm;
lanczos_eval_conv_thr	REAL	10^{-2}	Threshold for convergence of eigenvalue in Lanczos;
Controlling the eigenvector push			
eigval_thr	REAL	−0.01	Threshold for λ_{min} which determines when to start following the eigenvector;
eigen_step_size	REAL	0.2	Maximum size of the displacement with $\mathbf{V}_{\mathrm{min}}$;
eigenvec_guess	CHAR	" "	Filename where the eigenvector guess is read;
nsmooth	INT	0	Number of smoothing steps from initial displacement to eigenvector;
neigen	INT	1	Number of pushes along $\mathbf{V}_{\mathrm{min}}$ before starting a perpendicular relax;
nnewchance	INT	0	Number of times a research is allowed to re-initialize the first Lanczos vector;
Control the perpendicular relaxation			
nperp	INT	-1	Maximum number of perpendicular relaxation steps in all ARTn macro steps;
nperp_limitation	INT(:)	(4, 8, 12, 16, -1)	Limit of perpendicular relaxation steps for each ARTn step;
Controlling convergence			
forc_thr	REAL	10^{-3}	Final force convergence on both $\mathbf{F}_\perp$ and $\mathbf{F}$ threshold
converge_property	CHAR	"maxval"	Define which forces quantities are compared, the maximum value or the norm of the field. Possible values: "maxval" or "norm"
push_over	REAL	1.0	factor multiplied by eigen_step_size to push the configuration over the saddle point, before starting the final relaxation

Table 1: The input parameters of pARTn. Default values are in atomic units (lengths in bohr, energies in Ry, forces in Ry/bohr, and eigenvalues in Ry/bohr^2.