Computational Materials Science 163 (2019) 24–36

Contents lists available at ScienceDirect

Computational Materials Science

journal homepage: www.elsevier.com/locate/commatsci

pyiron: An integrated development environment for computational materials science

Jan Janssen[a,*], Sudarsan Surendralal[a], Yury Lysogorskiy[b], Mira Todorova[a], Tilmann Hickel[a], Ralf Drautz[b], Jörg Neugebauer[a]

[a] Max-Planck-Institut für Eisenforschung GmbH, Max-Planck-Str. 1, 40237 Düsseldorf, Germany
[b] Atomistic Modelling and Simulation, ICAMS, Ruhr-Universität Bochum, D-44801 Bochum, Germany

ARTICLE INFO

Keywords:
Modelling workflow
Integrated development environment
Complex simulation protocols

ABSTRACT

To support and accelerate the development of simulation protocols in atomistic modelling, we introduce an integrated development environment (IDE) for computational materials science called pyiron (http://pyiron.org). The pyiron IDE combines a web based source code editor, a job management system for build automation, and a hierarchical data management solution. The core components of the pyiron IDE are pyiron objects based on an abstract class, which links application structures such as atomistic structures, projects, jobs, simulation protocols and computing resources with persistent storage and an interactive user environment. The simulation protocols within the pyiron IDE are constructed using the Python programming language. To highlight key concepts of this tool as well as to demonstrate its ability to simplify the implementation and testing of simulation protocols we discuss two applications. In these examples we show how pyiron supports the whole life cycle of a typical simulation, seamlessly combines ab initio with empirical potential calculations, and how complex feedback loops can be implemented. While originally developed with focus on ab initio thermodynamics simulations, the concepts and implementation of pyiron are general thus allowing to employ it for a wide range of simulation topics.

1. Introduction

Over the past years, computational materials science has made tremendous progress in both predictive power and scalability, e.g. by high-throughput computations. These advances are not only related to the large gain in computer power but often to the development of advanced and computationally highly efficient algorithms and methods. A challenge in this respect is that the resulting simulation protocols are getting ever more complex: they often require the combination of specialised codes with incompatible input/output formats, the implementation of flexible interfaces to adjust the order of the steps in the simulation protocol dynamically and the distribution of tasks on heterogeneous computing platforms. This complexity hampers not only the development and implementation of new approaches but exceedingly slows down the dissemination of algorithms. One reason is that acquiring the necessary expertise and technical skills to actually use these simulation protocols often becomes too time consuming.

Over the past decade we systematically developed and explored fully ab initio based approaches to describe materials properties over the entire temperature window up to melting [1–3]. This required to accurately compute all relevant entropic contributions, such as electronic, vibronic, magnetic as well as their coupling [4–6]. To simultaneously achieve both the highest accuracy and efficiency we often had to combine codes developed by different scientific communities with very different input/output formats and concepts. To be more specific, for some of the approaches it was necessary to combine density-functional theory (DFT) calculations (VASP [7–9]) with empirical potential calculations (LAMMPS [10]) as well as spin quantum Monte Carlo approaches (ALPS [11]). To foster the development of these simulation protocols and their dissemination and use, we developed a framework that allows for the automation of routine tasks, stores all input/output data of large numbers of individual jobs in a generic format together with the simulation protocols and the parent–child relationship of the tasks. While originally targeting this framework only for the specific field of thermodynamic simulations, recent developments towards abstraction of the underlying concepts and collaborations with other groups made us realize that this tool is highly versatile and of interest to a broader community.

Similar abstraction layers have previously been built, starting with the atomic simulation environment (ASE) [12] and high throughput

* Corresponding author.
E-mail address: janssen@mpie.de (J. Janssen).

https://doi.org/10.1016/j.commatsci.2018.07.043
Received 7 May 2018; Received in revised form 17 July 2018; Accepted 21 July 2018
Available online 15 March 2019

frameworks such as AFLOW [13], the materials project infrastructure [14] with Fireworks [15], pymatgen [16] and ATOMATE [17] or AiiDA [18] and commercial frameworks like Material Studio [19], or MedeA [20]. Using one or several of these frameworks as basis more specialized tools have been developed. Examples are MAterials Simulation Toolkit (MAST) [21], Mpinterfaces [22] and jarvis [23,24] which are all based on pymatgen [16] and compatible with ASE [12]. Additionally, stand-alone tools that are not derived from any of these platforms exists. Examples are pylada [25] to setup and analyse defect calculations or openkim [26] and atomman [27] to provide user friendly interfaces for empirical potentials. The aim of all these tools is to enhance the researcher's productivity by automating the execution of individual calculations and implementing best practice workflows to accelerate the learning curve of the otherwise complex, error prone and labor-intensive steps within the different simulation protocols. Starting from a very different simulation task – implementing and running simulation protocols for thermodynamic calculations that have to handle large amounts of data such as trajectories for each run – we came up with solutions that are rather different from the existing frameworks. With this paper we want to describe and discuss the underlying concepts and design criteria. To illustrate these concepts we provide code examples of how to use pyiron.

2. Simulation life cycle

2.1. Example application

To illustrate the following considerations and to motivate the specific objectives of our approach we first consider a simple example: The computation of equilibrium values of parameters such as lattice constant and bulk modulus of a perfect, i.e. defect free, single crystal. This is commonly done by computing the energy volume (E-V) dependence at discrete points and fitting the dependence to a suitable analytical expression. A popular analytical model that is used for such a fit is the equation of state proposed by Murnaghan [28]:

$$E(V) = E_0 + B_0 V_0 \left[\frac{1}{B_0'(B_0' - 1)} \left(\frac{V}{V_0} \right)^{1-B_0'} + \frac{1}{B_0'} \frac{V}{V_0} - \frac{1}{B_0' - 1} \right].$$

In this equation E_0 is the equilibrium energy, B_0 the bulk modulus at the equilibrium volume V_0 and B_0' its first derivative. These four quantities are used as parameters to obtain the best fit through the computed $E_i(V_i)$ data points. While this example is simple and commonly runs manually or by a few lines of shell script commands, it has all the key aspects of more complex simulations. We will therefore use it in the following to discuss the fundamental steps of a typical simulation and analysis setup.

2.2. Simulation life cycle

A simulation life cycle such as the calculation of the E-V dependence consists of three stages: (i) input by the scientist (model, parameters, etc.), (ii) running the simulation, and (iii) analysis/visualization of the data. Often, such a cycle does not consist only of a single loop. Depending on the analysis in stage (iii) additional tasks may be formulated. With this new data the cycle is run repeatedly. The simulation life cycle may be divided in twelve steps as shown in Fig. 1. The individual steps are:

Step 1. The simulation life cycle starts with defining a model and a suitable set of initial parameters. This step is usually less formalized but important for the consistency of the whole cycle. For the specific example discussed here (E-V dependence) parameter choices are the initial lattice constant as well as the number and interval of sample volumes V_i.

Step 2. Based on the model a project is defined as a set of instructions specifying which simulation jobs have to be executed. In the simplified case of Fig. 1 instructions for a single calculation are defined. In case of an E-V dependence a common choice is to calculate the total energies E_i for the atomistic structures of 5–15 volumes V_i, spanning an interval of up to ± 10% around the initial guess for the equilibrium volume.

Step 3. The definition of the project generally has a generic part that is independent of the specific code used for the total energy calculations. For the specific example these relate to constant volume calculations and relaxation of the atoms.

Step 4. To run the actual total energy calculations the generic input is translated into a code specific input format. This input together with other inputs such as the employed potentials are communicated to a total energy code running in a working directory. For example, using VASP [7–9] for each of the volume calculations a POSCAR file needs to be created.

Step 5. The simulation code is executed in the working directory. Some simulation codes offer interfaces or scripting languages for communications during the execution, while others require input and output files.

Step 6. After the simulation has finished successfully the code specific output needs to be parsed to obtain the input parameters for the model. For the example discussed here these are the last (converged) energies E_i for each considered volume V_i.

Step 7. To analyse the results the output is often translated to a more generic output format, e.g. converting the units or rescaling the parameters. For the present example with a single species, it is often convenient to convert to per atom values. This allows to compare different crystal structures using a Murnaghan fit.

Step 8. With the results in a generic format the job is analysed to check its consistency, e.g. by validating convergence criteria of the individual simulation jobs.

Step 9. If all jobs within a project converged and prove to be consistent, the results of the individual jobs are combined for further analysis. For the E-V dependence a list of volume energy pairs (E_i, V_i) is compiled.

Step10. The compiled data is analysed. For our example this would be fitting the energy volume pairs E_i, V_i to the energy volume curve using the Murnaghan equation.

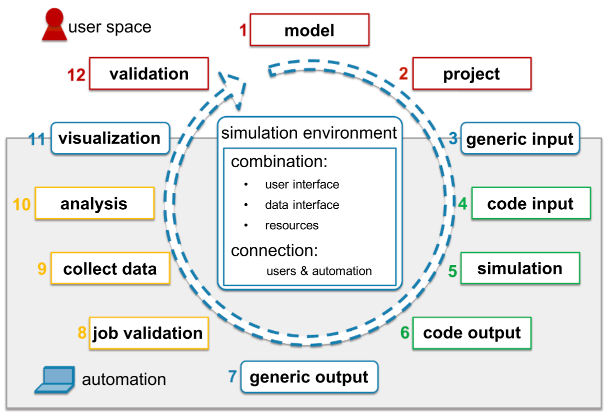


Fig. 1. Schematic view of a typical simulation life cycle, which illustrates the user's interaction with the various simulation tasks. It is divided in twelve steps, from the definition of the model – step 1 – to its validation – step 12. Three of these steps require obligatory input from the user, the model definition – step 1, the project definition – step 2, and the validation – step 12, while the rest can be automated. In particular the parsing of input and output data – steps 4 & 6 – as well as the collection of data – step 9 – are common tasks to any simulation. Using an intermediate layer with generic input – step 3, generic output – step 7 – and generalized tools for visualization – step 11, provides an abstraction layer that is code independent.

Step 11. The results are visualized to allow the user to analyze and discuss the simulation data.

Step 12. Finally, the results of steps 10 and 11 are validated by comparing them to other results or experimental data. Based on this validation the model hypothesis is confirmed or rejected. If the available data is insufficient a new set of calculations is run. In the example of the E-V dependence this would be the case if the initial guess turns out to be too far from the computed equilibrium volume, since an asymmetric distribution of points around the equilibrium can lead to large errors in the bulk modulus and its derivative.

3. pyiron

To support and assist with setting up the model and performing all the steps above we have developed a Python based framework called pyiron (http://pyiron.org). The key idea behind it is to provide a single tool with unified interface for all routine tasks. This allows to shield the user from the technicalities and specific input/output formats of the various codes and tools. While this is important for routine tasks, for specialized tasks or applications a critical design aspect is that the user always has fall-back options, i.e., to allow the user to access all or part of the job specific input/output directly or to give users all the freedom to explore new and not yet implemented routes.

The concept we follow is closely related to an Integrated Development Environment (IDE). Such IDEs exist for many programming languages and enable developers to focus on the project rather than on technicalities. In this respect we see pyiron as an integrated development environment for computational materials science (pyiron IDE).

3.1. pyiron objects

To accomplish the targets described above, we developed an abstract layer of objects (called pyiron objects). The key idea is to have a single base class that can be easily adopted to all the steps of the simulation protocol. Furthermore, it allows one to derive all the objects needed in a materials simulation environment such as atomic structures, projects, jobs, simulation protocols, or computing resources. To achieve usability, flexibility and automation – a pyiron object links (i) an easy to use and generic user interface (Section 3.1.1) with (ii) a data storage solution that is optimized with respect to the type of data commonly generated in materials science simulations (Section 3.1.2) and (iii) an interface to required resources (Section 3.1.3). The main objective is to provide the user with a universal and model independent interface that shields the user from the technicalities and complexity of the specific code as well as the underlying project and data structure. The internal structure of such an object is shown in Fig. 2. The individual parts are discussed in the following:

3.1.1. User interface

The pyiron object base class has been constructed to seamlessly “blend in” with existing Python modules and classes and to behave in a “Pythonic” manner [29]. This is achieved e.g. by operator overloading, allowing use of these objects as iterators, or by enabling complex index manipulations. Also, objects describing very different aspects of the simulation workflow appear to the user as behaving in the same way since they provide an identical syntax. We will highlight these design criteria in the example workflow discussed in Section 4.

In addition, the pyiron objects have been designed to make use of the features provided by Jupyter Notebooks [30]. Output such as project lists, queuing information regarding the job status etc. is given as a pandas object [31]. This has the advantage that it is rendered by Jupyter as formated table. Auto-completion and doc-strings provide navigation tools to explore and use the pyiron objects. Finally, atomic structures can be visualized and manipulated directly in the Jupyter

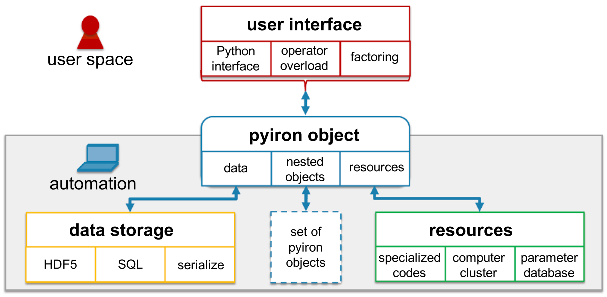


Fig. 2. Internal structure of a pyiron object. The pyiron objects provide a standardized interface to all layers of the data storage (HDF5, SQL-database, filesystem) and the resources (executables, queuing system, common input files). Each pyiron object can be the parent of lower level pyiron objects, which allows to construct complex and hierarchical workflows. The user interface of the pyiron object is standardized and behaves in many aspects like a native Python object.

notebook using the NGLview library [32]. Since the notebooks run as clients on any web-browser an easy to use client–server concept can be realized, where data production and analysis is performed directly on a remote cluster without forcing the user to copy the data to a local client. In a multi-user environment a Jupyterhub server can handle the access to the individual Jupyter notebooks.

3.1.2. Data storage

Data that is produced in materials simulations can be divided in two categories:

- Data that is independent of the specific simulation code used. Examples of this type are related to logging job specific data such as name and version of the executable used, date and time of submission, duration of the job, configuration of the computing resource, user name, number of cores, queue name, project name and path. This data is structured and can be efficiently stored in an SQL database.
- Data that is job specific (e.g. energy cutoff that applies to plane wave electronic structure codes but has no equivalent in an empirical potential calculation). This data type depends on the specific calculation or represents large multidimensional arrays (e.g. charge densities or MD trajectories). For this type of less structured data an SQL database is less suited.

The HDF5 format [33] is convenient to store the second type of rather unstructured data in computational materials science. It offers datasets for multidimensional arrays that link directly to Python’s numeric library *numpy* [34], hierarchical groups similar to the folders in a filesystem to store sparse data, and provides high-throughput I/O and compression with fast and nearly constant random access times to data subsets. For the same reasons the HDF5 format was also recommended as generalized format for MD trajectories in H5MD [35] and for the storage of volumetric data in pymatgen [16]. Particularly for large numerical datasets such as trajectories from MD simulations, HDF5 provides a significantly faster read and write access than JSON, which is popular in atomistic high throughput simulation platforms (s. Appendix B).

To accommodate these rather different needs of the various data types the data storage scheme of pyiron combines an SQL-database for the structured data with HDF5 files to store the unstructured simulation data. In addition pyiron objects enable access to the original input and output files. This combination of various storage formats allows for fast access, optimized memory due to compressed entries, and platform independence, but comes at the price of higher complexity than a solution based on a single database type. For the user the access is identical for all three storage types, i.e., the user does not need to know where the data is located. The access to the database, to the HDF5 or the input/output data files, including all pathnames, user accounts etc., is handled internally by the pyiron objects and is not exposed to the user unless specifically desired.

3.1.3. Resources

To interface with simulation codes as well as required resources (e.g. atomic potentials, information regarding the queuing system, working directories) system level interfaces are directly integrated in the pyiron objects. They can either be implemented by creating the required input files and executing the specialised code in a separate sub-process or by leveraging Python bindings or other interfaces if available. The aim is again to shield the user from the complexity of the underlying infrastructure. The resources interface provides a link to the parsers required to translate the generic input to the job specific input, the data necessary to run the job (e.g. atomic potentials) as well as to collect the output of the finished job and transfer it to the generic simulation code independent output. It also handles all aspects of the job execution, i.e., it submits the job to a queuing system, triggers the execution on the local machine or provides all information regarding the requested computer resources (number of cores, name of the queue, specific architecture or run time limits).

3.2. Functionality and interaction of the pyiron objects

pyiron objects have been designed to capture the interrelation of the individual steps in a simulation protocol. To achieve this, pyiron objects can either be linked in the SQL database, be nested inside each other, or both. As outlined in the following this has several advantages for the user.

3.2.1. Serialization

Python allows to serialize objects via the pickle file format protocol. Serialization means here that all information of the object, including its present state, is stored in a cross-platform file format. This can be used to suspend a job or session and to retrieve it later. To make full use of the HDF5 format used by the pyiron objects, we have implemented a serialization in pyiron based on this format. Again, to shield the users from the underlying complexity, in practice they will never have to call the serialization routines. Rather, they are natively implemented in the higher level commands such as the `run()` command which starts a job as demonstrated in the example calculation in Section 4.3. By calling this command all input, status, inheritance, and output information of the jobs are stored in HDF5. This feature is called orthogonal or transparent persistence [36], as the object state outlives the process it was created in.

The database links all the pyiron objects by providing a unique identifier (database index), the path where the HDF5 object is located, and the parent object that created it. Since the simulation protocols and the jobs are run within pyiron all this information is available and entered automatically, i.e. without any user input, into the database.

3.2.2. Orthogonal persistence

Orthogonal persistence is integrated in each pyiron object and is a core feature of the framework. It allows objects to get suspended and be continued by the user or other pyiron objects. For example, a job series creates child objects, submits them into the job scheduler queue and suspends itself. When one of the child objects is finished it continues the execution of the job series object to collect the output data, to perform analysis over all child jobs or to start new child jobs using the input of the finished ones. This functionality is at the heart of the job submission and control system implemented in pyiron and is discussed in Section 5.1. In addition the orthogonal persistence allows to move, copy or inspect pyiron objects and their meta data in a consistent way.

3.2.3. Factory pattern

To ensure a continuous automated data flow, pyiron objects inherit all relevant information from the initiating object, i.e., the one which created them instead of creating them from their corresponding classes. The advantage for the user is that the creation of objects that represent the next step in the simulation life cycle can be done directly from the existing object, i.e., without having to import the corresponding module, create an object and assign data to it. Thus, the derived object contains a link to the parent object, allowing to go from any element/object of the simulation life cycle to the parent or child objects.

The intrinsic link between all pyiron objects accelerates the development and testing of simulation protocols. In the following we will give examples to demonstrate how these objects allow the realization of the simulation life cycle described in the Section 2.

4. Example of a pyiron workflow

4.1. Creation of a project

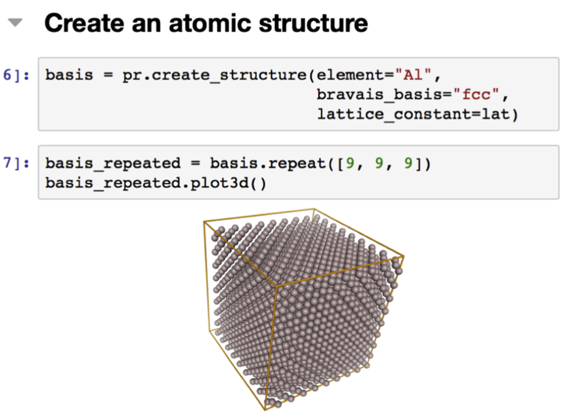
After the installation of pyiron, which is explained in Appendix A, the following code example can be executed. As a first step pyiron is imported, as well as further Python modules such as e.g. numpy [34] or matplotlib [37] as needed. Then a (pyiron) project object instance is created that takes as parameter the relative or absolute path of a new or existing project:

```
from pyiron import Project
pr = Project(path="projectname")
```

The creation of the project object instance provides the user with an interface to the file system and the database. A relative path name as given above starts from the local directory where the script or Jupyter notebook is located. Having the link to the database and the file system allows the user to inspect the content of the project. Examples are commands like:

```
pr.list_groups()
> > > [] # ["sub_project_a",...]
pr.list_nodes()
> > > [] # ["calculation_1", ...]
pr.path
> > > "/pyiron_projects/projectname"
```

These commands list the subprojects (groups) in the project path, pyiron objects (nodes) such as running or finished jobs created within this project and the absolute path name where the input, output and HDF5-files belonging to these objects are located in the filesystem. On the filesystem level a group would be a folder and a node would be a file, while on the HDF5-file level a group would be an HDF5 group and correspondingly a node would be an HDF5 node. Since so far our project is empty the list of groups and nodes is empty. The path is typically not needed by the user but shown here for completeness. The commands also demonstrate how a pyiron object connects the various storage types, i.e., the SQL database, HDF5-files and the conventional file system without requiring the user to know about the underlying complexity as discussed in Section 3.1.2.



Create an atomic structure

```
In [6]: basis = pr.create_structure(element="Al",
                                    bravais_basis="fcc",
                                    lattice_constant=lat)

In [7]: basis_repeated = basis.repeat([9, 9, 9])
        basis_repeated.plot3d()
```

Fig. 3. Snapshot of a Jupyter notebook running pyiron. The example shows the creation of a structure object, its manipulation (extension of the unit cell to a $9 \times 9 \times 9$ supercell) and its visualization.

4.2. *Create an atomic structure*

For an atomistic simulation the next pyiron object commonly created is the atomic structure. As a simple example we chose here an aluminium (Al) face-centered cubic (fcc) cell with lattice constant `lat`:

```
lat = 4.05
basis = pr.create_structure(element="Al",
                 bravais_basis="fcc",
                 lattice_constant=lat)
```

Note that the new pyiron object is created from the existing project object instance `pr` rather than importing a pyiron structure class and initializing it. This approach to create new pyiron objects from existing ones – factory pattern – is an integral part of any pyiron object (Section 3.2.3) and has two advantages: The user can use auto-completion (which is supported by most Python editors) but more importantly all relevant information can be copied from the parent to the child object maintaining the object hierarchy. This guarantees that all relevant metadata describing e.g. the simulation protocol, the location of the project directories or of the database are available and stored without forcing the user to take care of these issues. Once the structure object has been created it can be manipulated and visualized:

```
basis_repeated = basis.repeat([9,9,9])
basis_repeated.plot3d()
```

The first command extends the system to a $9 \times 9 \times 9$ supercell, the second plots an interactive 3d plot in the notebook using NGLview [32]. Fig. 3 illustrates how the above pyiron commands are run within a Jupyter notebook. To support more complex structures and structure manipulations, pyiron offers an interface to the structure class of ASE [12].

4.3. *Create and run a MD simulation*

As a next step we set up a VASP [7–9] calculation, assign the previously created structure and run an ab initio molecular dynamics simulation at 800 K.

```
job = pr.create_job(job_type=pr.job_type.Vasp,
                    job_name="job_vasp")
job.calc_md(temperature=800.0)
job.structure = basis
job.run()
> > > The job job_vasp was saved and
      received the ID: 1
```

The first line creates a VASP job (note that the job type is derived from the pyiron project object allowing auto-completion). Since it is derived from the project object it has all the information of the project and the corresponding file paths. Therefore the only additional information that has to be provided is a unique `job_name` as identifier inside the project. In the next line the calculation type is specified. `calc_md()` stands here for a molecular dynamics calculation and the parameter temperature specifies a value of 800 K. The following line assigns the structure object created before to the job object. In combination with the creation of the structure – Section 4.2 and the definition of the project – Section 4.1 – these represent steps 1–3 in the simulation life cycle of an individual calculation.

Finally the `run()` command executes the calculation. In terms of the simulation life cycle this relates to steps 4–10, illustrated by the grey background in Fig. 1. In addition to VASP and LAMMPS, which are currently supported, additional simulation codes can be interfaced by the user as described in Appendix C.

Note that the pyiron script used to set up and run the simulation is generic. Specifically, it does not require to provide any VASP specific input data. If modifications of the VASP input are required, e.g. to modify the plane wave basis energy cutoff, the corresponding information can be readily provided:

```
job.input.incar["ENCUT"] = 300.0
```

To inspect the change or to see the default parameters all input files print their content as pandas style formatted tables [31]. For a VASP job running a molecular dynamics calculation at 800 K and with the above modified `"ENCUT"` parameter the input parameters are printed as:

```
job.input.incar
> > >               Parameter        Value          Comment
> > > 0             SYSTEM           job_vasp       jobname
> > > 1             PREC             Accurate
> > > 2             ALGO             Fast
> > > 3             ENCUT            300.0
> > > 4             LREAL            False
> > > 5             LWAVE            False
> > > 6             LORBIT           0
> > > 7             SMASS            3
> > > 8             TEBEG            800.0
> > > 9             NSW              1000
> > > 10            NBLOCK           1
> > > 11            POTIM            1
```

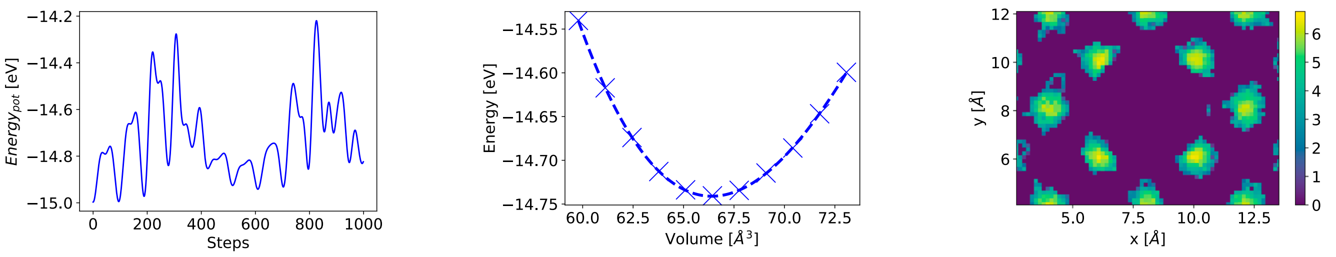
Instead of setting the VASP specific `"ENCUT"` parameter in the INCAR file, pyiron provides generic commands for a large set of input parameters e.g.:

```
job.set_encut(300.0)
```

pyiron assigns them not only a generic name but also converts the units. In general, all energy units are given in eV, temperatures in K and coordinates in Å. An advantage of this generic notation is that these commands do not have to be changed when switching to another plane wave code. Even switching from an ab initio plane wave code such as VASP to an empirical potential code such as LAMMPS requires only minor changes:

```
job = pr.create_job(job_type=pr.job_type.Lammps,
                    job_name="job_lammps")
job.calc_md(temperature=800.0)
job.potential = "Al_H_Ni_Angelo_eam"
job.structure = basis
job.run()
> > > The job job_lammps was saved and
      received the ID: 2
```

Note that one has to change only the `job_type` and provide the name of the specific empirical potential. For a VASP job the xc-



(a) Potential energy (b) Energy Volume curve (c) 2D-pair correlation function

Fig. 4. Examples of data analysis and visualization in pyiron (a) potential energy vs. time step from a molecular dynamics simulation (Section 4.6). (b) Plot of energy versus volume (Section 5.1). (c) Analysis of the local anharmonicity as described in Ref. [38] (Appendix E).

functional (PBE) and the recommended pseudopotential for the chosen species is selected by default. For such a case there is no need to provide the potential explicitly. However, such recommendations are not available for empirical potentials. Therefore, once the `job_type` and atomic structure are defined, a list of available potentials for a given atomic structure can be obtained and selected by:

```
job.list_potentials()
>>> ["Al_H_Ni_Angelo_eam",
     "Al_Mg_Mendelev_eam",
     "Al_Ti_Mishin_eam"]
job.potential = job.list_potentials()[0]
```

In this case, from the list of available potentials the aluminium-hydrogen-nickel potential from Angelo [39] is selected. A full list of the available potentials and their corresponding parametrisations is available as pandas style formatted table with the `view_potentials()` command.

4.4. Define computing resources

pyiron offers an interface for calculations that require more computing resources. For example, to enable multiprocessing on a single computing resource one simply specifies the number of computing cores:

```
job.server.cores = 8
```

pyiron objects also offer various modes of execution of the underlying specialised codes. The default execution mode is modal – foreground execution. In this mode the pyiron object waits for the specialized code (e.g. LAMMPS) to execute a given task before it continues. Another one is the non-modal mode – background execution – which enables asynchronous communication. In this mode the pyiron object continues operation while the specialized code is executed in a separate subprocess – Appendix D. The asynchronous mode allows e.g. to submit the pyiron object to a job scheduler:

```
job.server.list_queues()
>>> ["queue_small", "queue_big"]
job.server.queue = "queue_small"
```

The first command lists the available queues. With the second command the user specifies one of the available queues by name. A full list of the available queues and their configuration can be obtained as pandas style formatted table via `view_queues()`

4.5. Inspect the project

The design of the pyiron objects gives access to all data in the project. To manually navigate through the project we type the project object instance `pr`:

```
pr
>>> {"groups": [],
     "nodes": ["job_vasp", "job_lammps"]}
```

This command is a combination of the `list_nodes()` and `list_groups()` functions introduced in Section 4.1. It lists the two jobs calculated within the project. To see the content of one of the jobs we can specify:

```
pr["job_vasp"]
>>> {"groups": ["output", "input"],
     "nodes": ["server", "NAME",
               "TYPE", "VERSION"]}
```

4.6. Analyse the project

To analyze, visualize and validate the calculations – steps 10-12 in the simulation life cycle (Section 2) – the above described approach can be employed iteratively. The design of the pyiron objects makes it easy to access data on lower lying levels. For example, to analyze the potential energy from the previously performed MD runs we write:

```
e_pot = pr["job_vasp/output/generic/energy_pot"]
steps = pr["job_vasp/output/generic/steps"]
```

Here it is again seen how the user is shielded from the actual physical storage: `"job_vasp"` defines an HDF5 file and `"/output/generic/energy_pot"` the path within the HDF5 file.

The above output returns numpy arrays [34] that can be processed further. The output, just like the input, is converted by assigning generic names and units. Thus, `"energy_pot"` gives always a numpy array containing the sequence of potential energies in eV obtained during the simulation.

Since all the output can be accessed via numpy arrays it can be processed and visualized using standard Python modules. For example, to plot the energy versus time steps we can use the matplotlib library:

```
import matplotlib.pyplot as plt
plt.plot(steps, e_pot)
plt.xlabel("Steps")
plt.ylabel("$Energy_{pot}$ [eV]")
```

The resulting graph is plotted in Fig. 4a. Atomic positions, forces, temperature, pressure etc. can be accessed in the same way. To give a specific example, based on the atomic positions the effect of local anharmonicity can be graphically analyzed. The employed algorithm is described in Ref. [38]. The resulting graph is plotted in Fig. 4c and the pyiron code given in Appendix E.

A major advantage of storing the output in a generic format, both with respect to the name space and the physical units is that we have only to modify the name of the job when analysing a calculation performed by another code. For example, to get the potential energy from the previous LAMMPS calculation we write:

```
e_pot = pr["job_lammps/output/generic/energy_pot"]
```

pyiron also provides tools to iteratively walk through the project. To plot e.g. all MD energy curves from a project in a single plot we can write:

```
for job in pr.iter_jobs():
    e_pot = job["output/generic/energy_pot"]
    plt.plot(e_pot, label=job.name)
plt.legend()
```

Note that the analysis script makes no assumptions regarding the type of job, i.e., whether it is a LAMMPS or VASP job. Thus, scripts that analyze energies, trajectories etc. are fully reusable and can be applied to any atomistic simulation code.

pyiron offers a fast inspect mode that allows to return individual entries inside the HDF5 file with low latency. In this mode only the specified subgroup is read in rather than creating a full object. This mode is activated by setting the parameter `path="output/generic"` in the `iter_jobs()` function. The underlying techniques are discussed in Section 3.1.2.

5. Application

To demonstrate how the pyiron concepts can help to accelerate the development of simulation protocols we show how efficiently a single calculation e.g. to compute a single E-V curve (Section 5.1) can be extended to a high throughput evaluation (Sections 5.2 and 5.3).

5.1. E-V dependence at 0 K

Based on the already provided code segments the code to calculate the E-V dependence at 0 K is:

```
import numpy as np
pr = Project("application")
job_ref = pr.create_job(job_type=pr.job_type.Vasp,
                        job_name="job_ref")
job.structure = basis
job.structure.set_relative()
for strain in np.linspace(0.9, 1.1, 11):
    job_strain = job_ref.copy()
    job_strain.name = "strain_" + str(strain)
    job_strain.structure.cell *= strain ** (1/3)
    job_strain.run()
```

As a first step a reference job is set up following the example given in Section 4. The atomistic structure is set to relative coordinates using `set_relative()`. This allows to apply strain by simply scaling the lattice constant. Specifically, eleven strain values ranging from 90% to 110% are used. The individual jobs are created by copying the reference job, renaming, straining and executing it. The energy volume curves computed in this project are analysed by:

```
vol, erg = [], []
for job in pr.iter_jobs (path="output/generic"):
     vol.append(job["volume"])
     erg.append(job["energy_tot"][-1])
plt.plot(vol, erg)
plt.xlabel("Volume [$\\AA^3$]")
plt.ylabel("Energy [eV]")
```

The energy volume pairs V_i, E_i are collected in the lists `vol` and `erg`. The collected data is used to plot the resulting energy volume curve employing `matplotlib` (Fig. 4b). The option `path="output/generic"` not only shortens the path name when accessing data but also automatically switches to the fast reading mode. As pointed out in Section 4.6 this speeds up the data access by an order of magnitude.

Creating a set of jobs using a for loop is only one option. For many common tasks pyiron provides iterative job generators. Job generators can process jobs either in a serial or parallel manner and belong to the group of metajobs. For the user these jobs behave like regular jobs. The presently available types of metajobs are listed in Appendix F. Example metajobs, which can be generated with these types and are implemented in the initial release of pyiron are the evaluation of E-V curves and convergence checks with respect to planewave basis energy cutoff and k-points. Further metajobs will be published on the pyiron website [40]. We note, however, that a primary goal of pyiron is to provide users with all the tools needed to easily implement and test their own metajobs.

To show an application of a metajob the above code can be substantially shortened using the Murnaghan metajob:

```
murn = pr.create_job(job_type=pr.job_type.Murnaghan,
                     job_name="murnaghan")
murn.ref_job = job_ref
murn.run()
murn.plot()
```

Instead of using an explicit for loop the Murnaghan metajob performs automatically a large part of the simulation life cycle. It is created from the project object instance and the reference job is assigned as `ref_job`. The run command then creates and runs the individual jobs in parallel and collects the output of the individual subjobs. The metajob may contain specific analysis or plotting tools for further processing (last line). In the above example we use the default parameters for the E-V dependence. To be generic they are given in relative strains rather than lattice constants. Following the same notation as for generic pyiron jobs we can change the default input:

```
murn.input["vol_range"] = 0.05
```

Since the calculations for the different strain states can be run independently the metajob object `murn` is based on a ParallelMaster. Thus, by default all the calculations will be executed in parallel. To enable parallelization we increase the number of cores for the Murnaghan metajob and set the `run_mode` to `non_modal`:

```
murn.server.cores = 11
murn.server.run_mode.non_modal = True
```

If the number of cores selected for the metajob is smaller than the number of jobs that have to be executed by the ParallelMaster, the object behaves like a job scheduler that runs a job whenever the previous job is finished.

Metajobs have the same syntax as a generic pyiron job object. However, in contrast to these they allow to control and run multiple jobs or even multiple metajobs. This functionality makes metajobs highly versatile building blocks in constructing complex simulation protocols. The underlying mapping of the workflow to a pyiron object and finally to the HDF5 data structure is shown for the above example in Fig. 5.

5.2. E-V dependence at finite temperature

To show the flexibility of the concept of metajobs to construct complex simulation protocols, we consider a metajob of metajobs. For this purpose we extend the previous example by replacing static T = 0 K calculations with finite temperature ones. Specifically, we compute each point of the E-V dependence by averaging over an NVT ensemble. We enforce a predefined statistical accuracy by performing for each volume consecutive MD runs until a predefined error target in the thermodynamic average is reached. The corresponding code

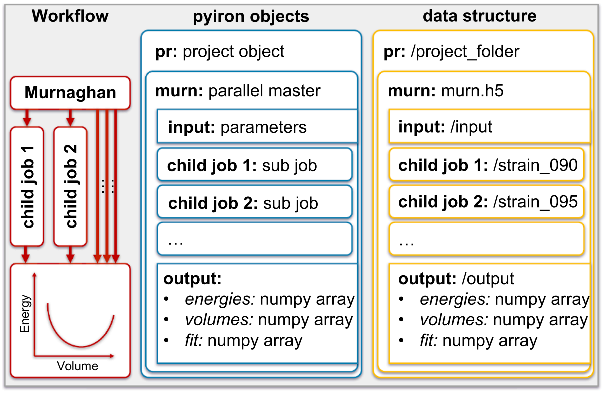


Fig. 5. Relation between workflow in the simulation protocol, pyiron object and data structure.

consists of a parallel Murnaghan master where each parallel thread consists of a SerialMaster. Only the SerialMaster consists of individual jobs:

```
def convergence_goal(self):
      import numpy as np
      eps = 0.1
      erg_lst = self.get_from_childs(
            "output/generic/energy_pot")
      erg_std = np.std(erg_lst)
      if erg_std / len(erg_lst) < eps:
            return True
     else:
            job_prev = self[-1]
            job_name = self.first_child_name()\
                            + "_" + str(len(self))
            job_next = job_prev.restart(
                            job_name=job_name)
            return job_next

job = pr.create_job(job_type=pr.job_type.Lammps,
                             job_name="lammps_md_run")
job.structure = basis
job.potential = "Al_H_Ni_Angelo_eam"
job.calc_md(temperature=800.0)
job_ser = pr.create_job(job_type=pr.job_type.SerialMaster,
                                   job_name="serial_master")
job_ser.ref_job = job
job_ser.set_goal (convergence_goal=convergence_goal)
murn = pr.create_job(job_type=pr.job_type.Murnaghan,
                              job_name="murnaghan_finite_temp")
murn.ref_job = job_ser
murn.run()
```

A new aspect in the above code is the possibility to assign to a pyiron object not only parameters but also new code. This is used to define the convergence criterion for the MD runs. The `convergence_goal()` function assigned to the SerialMaster creates MD jobs, until the potential energy time averaged threshold `eps=0.1` is reached. The SerialMaster containing the convergence goal and its reference job is then set as a reference job to construct the next higher level – the Murnaghan job. The concept of defining a hierarchy of serial, parallel and derived metajobs can be extended to arbitrary levels.

5.3. Evaluation of empirical potentials

The evaluation of empirical potentials is based on the comparison of the calculated atomistic properties to reference values. The reference can be computed data, e.g. calculated by ab initio methods (DFT) or have been obtained experimentally. Considering the large number of potential-property-prototype combinations a high-throughput approach with automatic calculation management is required. Therefore, the pyiron framework is used as a part of an automated empirical potential validation system [43], as it allows to apply the same simulation protocols for both VASP and LAMMPS.

In Fig. 6 one can see a typical set of E-V dependencies and their related properties – equilibrium energies E_0, volumes V_0, bulk moduli B_0 and B_0' for molybdenum. These calculations were performed by an EAM potential [41] and DFT-PBE [42]. Despite the large differences in the code and the methods, the identical simulation life cycle could be used. This makes an automated and validated comparison of the high-throughput data possible. Moreover, the job handling capabilities of the

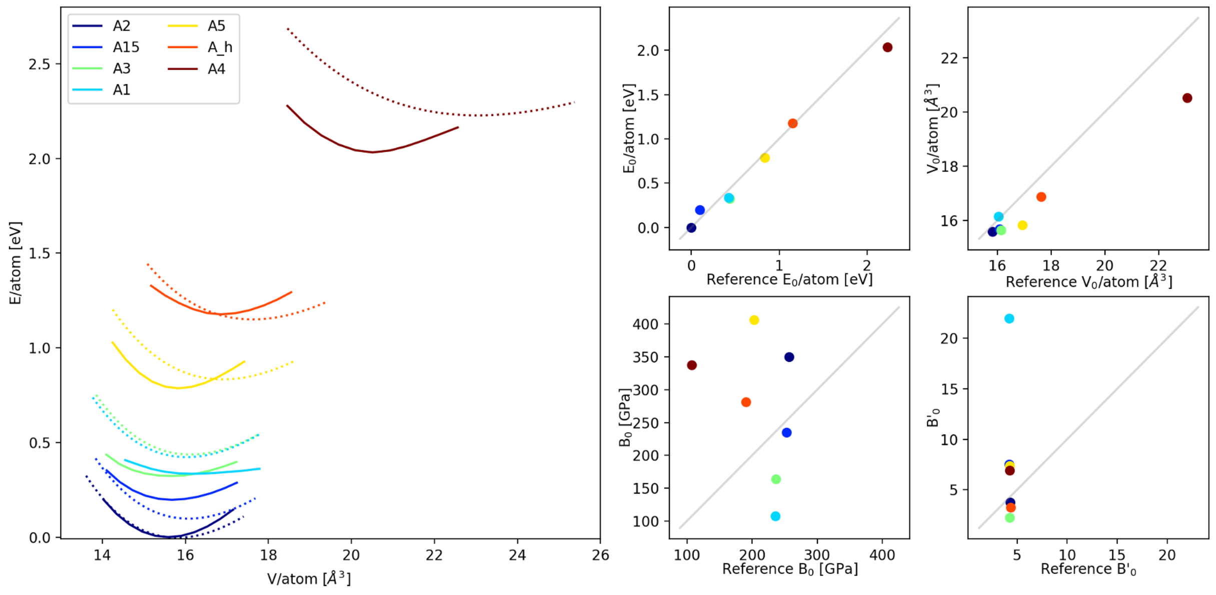


Fig. 6. Energy volume curves for five prototypical structures (A1-fcc, A2-bcc, A3-hcp, A4-diamond-like, A5-beta-Sn and A_h simple cubic structure) and related properties: Equilibrium energy E_0, volume V_0, bulk modulus B_0 and B_0': The quantities are calculated by an EAM potential for molybdenum [41] (solid lines) and compared to DFT-PBE reference calculations [42] (dotted lines) and the color of the dots in the right corresponds to the phase shown in the left.

pyiron framework allowed to schedule the empirical potential calculations on a single workstation. In contrast, the high performance computer cluster was used for the numerically expensive DFT calculations, without having to modify the simulation life cycle to upscale and utilize different computational resources.

6. Conclusions

The Python based framework pyiron presented here provides all the tools necessary to interactively implement and test simulation protocols and to upscale them for high-throughput simulations on large computer clusters. Like an integrated development environment (IDE) for programming languages it allows to perform all the steps necessary for a simulation life cycle within the same framework.

A key feature of pyiron are pyiron objects. This abstract object class is the origin of all derived user accessible tools such as projects, jobs, metajobs or complete simulation protocols. pyiron objects are characterized by a unified syntax and provide easy access to the complex data structure and the resources. They can also create new pyiron objects (factory pattern) or can contain groups of children pyiron objects. This, together with the ability to serialize and deserialize these objects in HDF5 allows to construct hierarchical simulation life cycles over several levels. Since the pyiron objects contain by construction all information regarding their parent as well as their child projects common tasks such as storing, copying or running a workflow are fully automated.

While pyiron can be run in any Python shell it has been designed to heavily use the features of Jupyter notebooks. Specifically, using it with a Jupyterhub [44] provides a powerful server-client system: the user can work with a standard web browser on a local computer to develop, run and analyze simulations in a remote computer center. The examples discussed here are meant as simple demonstrators. They show key concepts of pyiron as well as typical workflows and give a brief overview over the syntax. More complex and “real-world” examples can be found in Refs. [43,40].

Acknowledgement

The authors thank Christoph Freysoldt from MPIE, Ugur Aydin former MPIE member and Thomas Hammerschmidt from ICAMS for stimulating discussions as well as the team from the Max Planck Computing and Data Facility (MPCDF), namley Christian Guggenberger, Markus Rampp, Klaus Reuter and Thomas Zastrow for technical support. We acknowledge the Max-Planck Research network on “Big-data-driven Material science” (BigMax) for financial support. Finally, we thank the “Computational Materials Design” department at the MPIE and the “Atomistic Simulation of Structural and Phase Stability” group at ICAMS for testing pyiron and providing feedback.

Appendix A. Installation of pyiron

pyiron can be installed either via anaconda, via pip or from source. A full documentation is provided on the pyiron website [40]. An anaconda based installation is recommended:

```
conda install -c conda-forge pyiron
```

During the initial import pyiron asks to create the `.pyiron` configuration file and the resource directory:

```
import pyiron
>>> "It appears that pyiron is not yet configured, do you want to\
      create a default start configuration (recommended: yes).
      [yes/no]:"
```

The `.pyiron` configuration file allows pyiron to locate the resource directories stored in `RESOURCE_PATHS`, containing e.g. potentials and executables, as well as the `PROJECT_PATHS`, which are the directories pyiron projects can be created in. Both variables can contain multiple directories. The default `.pyiron` configuration files include:

```
[DEFAULT]
PROJECT_PATHS = ~/pyiron/projects
RESOURCE_PATHS = ~/pyiron/resources
```

The resource directories for example provide an easy way to select executables, by specifying a version rather than the full path to the executable:

```
job = pr.create_job(job_type=pr.job_type.Vasp, job_name="vasp")
job.version = "5.4.4"
job.executable.executable_path
>>> "~/pyiron/resources/vasp/bin/run_vasp_5.4.4.sh"
```

The required executables of VASP are not included in the pyiron IDE and have to be licensed and installed by the user. Next to the executables the resource directory contains additional files like the VASP pseudopotentials (also not included in the pyiron IDE). Those are located in:

```
"~/pyiron/resources/vasp/potentials"
```

The variable `RESOURCE_PATHS` is user-extendable, allowing the user to add new executable versions or additional pseudopotentials.

Appendix B. HDF5 performance

To compare the read and write performance of the JSON format and the HDF5 format we consider the trajectory of a system consisting of $n_A = 100$ atoms going over n_t time steps. This dataset is represented by a $n_t{\cdot}n_A{\cdot}3$ rank 3 tensor. As shown in Fig. B.7 even for short trajectories of only a few steps both the read and write performance of the HDF5 format are superior compared to JSON.

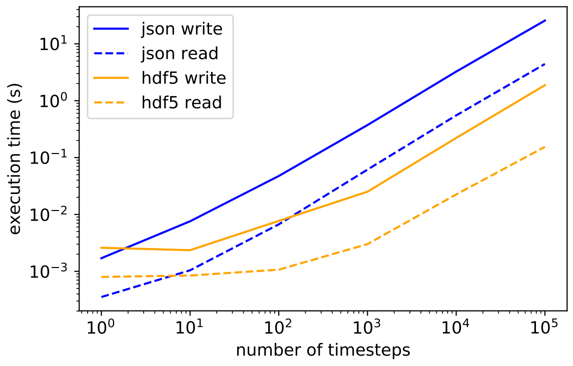


Fig. B.7. Comparing the read and write performance of HDF5 vs. JSON for handling atomic trajectories. The example system consists of 100 atoms, i.e., the size of the object is $3 \cdot 100 \cdot n$, with n being the number of time steps.

Appendix C. Implementing a new simulation code

The design criteria of pyiron objects allow to easily add new job classes and metajobs. Below we show all the necessary steps. We restrict ourselves on a simple toy example. To run it paste the following code into the file "`pyiron_example.py`". This file has to be in a directory contained in the `PYTHONPATH` variable.

```
from os.path import join
from pyiron.base.generic.parameters import GenericParameters
from pyiron.base.job.generic import GenericJob

class ToyJob(GenericJob):
    def __init__(self, project, job_name):
        super(ToyJob, self).__init__(project, job_name)
        self.input = ToyInput()
        self.executable = "cat input > output"

    def write_input(self):
        self.input.write_file(
            file_name="input",
            cwd=self.working_directory)

    def collect_output(self):
        file = join(self.working_directory, "output")
        with open(file) as f:
            line = f.readlines()[0]
            energy = float(line.split()[1])
        with self.project_hdf5.open(
              "output/generic") as h5out:
          h5out["energy_tot"] = energy

    def to_hdf(self, hdf=None, group_name=None):
        super(ToyJob, self).to_hdf(
            hdf=hdf,
            group_name=group_name)
        with self.project_hdf5.open("input") as h5in:
            self.input.to_hdf(h5in)

    def from_hdf(self, hdf=None, group_name=None):
        super(ToyJob, self).from_hdf(
          hdf=hdf,
          group_name=group_name)
        with self.project_hdf5.open("input") as h5in:
          self.input.from_hdf(h5in)
```

```
class ToyInput(GenericParameters):
    def __init__(self, input_file_name=None):
        super(ToyInput, self).__init__(
            input_file_name=input_file_name,
            table_name="input")

    def load_default(self):
        self.load_string("input_energy 100")
```

To keep the example simple we define with `self.executable` not a link to an actual simulation code as we normally would do. Rather we define the code here. This one line code simply copies the "`input`" into the "`output`" file. The `write_input()` creates the input file. `collect_output()` parses the output file into pyiron. `to_hdf()` and `from_hdf()` implement the serialisation into HDF5. The `ToyInput` class is derived from `GenericParameters`. Since this class is used for all inputs, a consistent user experience is provided. `load_default()` defines the default input. The thus implemented example job can be used like any other job type in pyiron:

```
from pyiron import Project
pr = Project('test')
job = pr.create_job(job_type=pr.job_type.ToyJob,
                    job_name="toy")
print(job.input)
>>>      Parameter Value Comment
>>>  0  input_energy 100
job.run()
>>> The job toy was saved and received the ID: 1
job['output/generic/energy_tot']
>>> 100.0
```

The implementation of more complex jobs and metajobs is explained on the pyiron website [40].

Appendix D. Run modes

pyiron allows to execute objects in various modes. Choosing the mode allows to easily switch between various tasks: interactive rapid prototype driven development, running large batches of production jobs or debugging pyiron based scripts. The modes are:

modal: The default run mode in pyiron. The Python code waits for the execution of the simulation code to finish before it continues. This mode of foreground execution is preferred for interactive development.

non_modal: Subprocesses are executed in the background. This mode should be used for medium sized jobs and if a queuing system is not available or needed.

queue: Distributes large calculations on a queuing system. This mode requires pyiron to be installed on the computer platform where the jobs are running. It also requires that all nodes have access to the central database. This mode enables complex metajobs to be executed in a fully decentralized manner. A central master job or daemon process is not needed.

manual: Generates the input files only. Allows to run executables such as LAMMPS or VASP manually but with pyiron generated input. This mode can be used to debug simulation protocols or executables such as LAMMPS or VASP.

The `run_mode` can be set in the server object:

```
job.server.run_mode.non_modal = True
```

The above syntax has been again designed to maximally utilize auto completion. The `run_mode` is set automatically to `queue` if a queue is selected:

```
job.server.queue = "queue_name"
```

Appendix E. Computing local anharmonicity

In the following we show a more complex example of analysing a MD simulation with pyiron. The example follows the approach introduced in Ref. [38] to compute the 2D-pair correlation function from the MD trajectories. Using pyiron these calculations and the corresponding analysis can be performed with the following lines of Python code. The resulting plot is shown in Fig. 4c.

```
import numpy as np
from matplotlib.colors import LogNorm
from pyiron import Project

pr = Project("anharmonic")
basis = pr.create_structure(element="Al",
           bravais_basis="fcc",
           lattice_constant=4.05)
basis = basis.repeat([4,4,4])
job = pr.create_job(job_type=pr.job_type.Lammps,
                    job_name="job_lammps")
job.calc_md(temperature=800.0,
            n_ionic_steps=10000,
            n_print=1)
job.structure = basis
job.potential = "Al_H_Ni_Angelo_eam"
job.run()

pos = job["output/generic/positions"]
alat = basis.cell[0,0]
bins = 100
xedges = np.linspace(0, alat, bins+1)
yedges = np.linspace(0, alat, bins+1)
X, Y = np.meshgrid(xedges, yedges)

H_sum = np.zeros([bins, bins])
for i_atom in np.arange(len(basis)):
    d_pos = np.transpose(pos, (1,0,2)) - pos[:, i_atom, :]
    d_pos= d_pos.reshape(-1,3)
    d_pos = np.mod(d_pos-alat/2, alat)
    d_ind = np.abs(d_pos[:, 2]-alat/2) < 1
    d_pos = d_pos[d_ind]
    H, xedges, yedges = np.histogram2d(
           d_pos[:, 0],
           d_pos[:, 1],
           bins=(xedges, yedges))
    H[int(bins/2), int(bins/2)] = 0
    H_sum += H

    plt.pcolor(X, Y, np.log(H+1));
plt.xlim([alat/2 - 4, alat/2 + 4])
plt.ylim([alat/2 - 4, alat/2 + 4])
plt.colorbar()
plt.axes().set_aspect("equal", "datalim");
```

Appendix F. Metajobs

The pyiron IDE includes four different types of metajobs. These are listed in Table F.1.

Table F.1
Different types of metajobs implemented in pyiron.

	Simple metajobs	Generators
Serial	ScriptJob	SerialMaster
Parallel	ListMaster	ParallelMaster

ScriptJob:	The simplest way to go beyond a single Jupyter notebook. The ScriptJob takes a Jupyter notebook as input and executes it in a separate sub-process. In particular in the experimental phase, this functionality allows to quickly reuse existing code and combine it, while maintaining the serial dependency of the individual jobs within the Jupyter notebook.
ListMaster:	Use it if there is no dependency of the individual jobs on each other. The ListMaster behaves like a Python list object, where the user can append multiple job objects. The jobs are distributed on the computing resources. If the number of jobs exceeds the number of parallel threats they wait until finished jobs free the resources.
ParallelMaster:	Similar to the ListMaster the ParallelMaster allows executing multiple jobs in parallel. In contrast to the former, the ParallelMaster contains a `create_next()` function. This function provides the rules how to automatically generate the jobs in the master.
SerialMaster:	Like the ParallelMaster it generates jobs using a `create_next()` function. In contrast to the ParallelMaster the new job can contain information from the previous one. It thus is executed in serial rather than in parallel. The SerialMaster may contain a function to validate a user defined convergence goal and stop generating new children as soon as the goal is reached.

This set of metajobs gives the users the ability to scale their simulation protocols beyond a single Jupyter notebook. The ability to nest and combine these metajobs allows to construct protocols with arbitrary complexity.

References

[1] T. Hickel, B. Grabowski, F. Körmann, J. Neugebauer, Advancing density functional theory to finite temperatures: methods and applications in steel design, J. Phys.: Condens. Matter 24 (5) (2012) 053202, https://doi.org/10.1088/0953-8984/24/5/053202 http://stacks.iop.org/0953-8984/24/i=5/a=053202.

[2] B. Grabowski, L. Ismer, T. Hickel, J. Neugebauer, Ab initio up to the melting point: Anharmonicity and vacancies in aluminum, Phys. Rev. B 79 (2009) 134106, https://doi.org/10.1103/PhysRevB.79.134106 https://link.aps.org/doi/10.1103/PhysRevB.79.134106.

[3] L.-F. Zhu, B. Grabowski, J. Neugebauer, Efficient approach to compute melting properties fully from ab initio with application to Cu, Phys. Rev. B 96 (2017) 224202, https://doi.org/10.1103/PhysRevB.96.224202 https://link.aps.org/doi/10.1103/PhysRevB.96.224202.

[4] B. Grabowski, P. Söderlind, T. Hickel, J. Neugebauer, Temperature-driven phase transitions from first principles including all relevant excitations: The fcc-to-bcc transition in Ca, Phys. Rev. B 84 (2011) 214107, https://doi.org/10.1103/PhysRevB.84.214107 https://link.aps.org/doi/10.1103/PhysRevB.84.214107.

[5] A.I. Duff, T. Davey, D. Korbmacher, A. Glensk, B. Grabowski, J. Neugebauer, M.W. Finnis, Improved method of calculating ab initio high-temperature thermodynamic properties with application to ZrC, Phys. Rev. B 91 (2015) 214311, https://doi.org/10.1103/PhysRevB.91.214311 https://link.aps.org/doi/10.1103/PhysRevB.91.214311.

[6] Y. Gong, B. Grabowski, A. Glensk, F. Körmann, J. Neugebauer, R.C. Reed, Temperature dependence of the gibbs energy of vacancy formation of fcc Ni, Phys. Rev. B 97 (2018) 214106, https://doi.org/10.1103/PhysRevB.97.214106 https://link.aps.org/doi/10.1103/PhysRevB.97.214106.

[7] G. Kresse, J. Hafner, Ab initio molecular dynamics for liquid metals, Phys. Rev. B 47 (1993) 558–561, https://doi.org/10.1103/PhysRevB.47.558 https://link.aps.org/doi/10.1103/PhysRevB.47.558.

[8] G. Kresse, J. Furthmüller, Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set, Comput. Mater. Sci. 6 (1) (1996) 15–50, https://doi.org/10.1016/0927-0256(96)00008-0 http://www.sciencedirect.com/science/article/pii/0927025696000080.

[9] G. Kresse, J. Furthmüller, Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set, Phys. Rev. B 54 (1996) 11169–11186, https://doi.org/10.1103/PhysRevB.54.11169 https://link.aps.org/doi/10.1103/PhysRevB.54.11169.

[10] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1) (1995) 1–19, https://doi.org/10.1006/jcph.1995.1039 http://www.sciencedirect.com/science/article/pii/S002199918571039X.

[11] B. Bauer, L.D. Carr, H.G. Evertz, A. Feiguin, J. Freire, S. Fuchs, L. Gamper, J. Gukelberger, E. Gull, S. Guertler, A. Hehn, R. Igarashi, S.V. Isakov, D. Koop, P.N. Ma, P. Mates, H. Matsuo, O. Parcollet, G. Pawłowski, J.D. Picon, L. Pollet, E. Santos, V.W. Scarola, U. Schollwöck, C. Silva, B. Surer, S. Todo, S. Trebst, M. Troyer, M.L. Wall, P. Werner, S. Wessel, The ALPS project release 2.0: open source software for strongly correlated systems, J. Statist. Mech.: Theory Exp. 2011 (05) (2011) P05001, http://stacks.iop.org/1742-5468/2011/i=05/a=P05001.

[12] A.H. Larsen, J.J. Mortensen, J. Blomqvist, I.E. Castelli, R. Christensen, M. Dułak, J. Friis, M.N. Groves, B. Hammer, C. Hargus, E.D. Hermes, P.C. Jennings, P.B. Jensen, J. Kermode, J.R. Kitchin, E.L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J.B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K.S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, K.W. Jacobsen, The atomic simulation environment-a python library for working with atoms, J. Phys.: Condens. Matter 29 (27) (2017) 273002, http://stacks.iop.org/0953-8984/29/i=27/a=273002.

[13] S. Curtarolo, W. Setyawan, G.L. Hart, M. Jahnatek, R.V. Chepulskii, R.H. Taylor, S. Wang, J. Xue, K. Yang, O. Levy, M.J. Mehl, H.T. Stokes, D.O. Demchenko, D. Morgan, AFLOW: An automatic framework for high-throughput materials discovery, Comput. Mater. Sci. 58 (2012) 218–226, https://doi.org/10.1016/j.commatsci.2012.02.005 http://www.sciencedirect.com/science/article/pii/S0927025612000717.

[14] A. Jain, G. Hautier, C.J. Moore, S.P. Ong, C.C. Fischer, T. Mueller, K.A. Persson, G. Ceder, A high-throughput infrastructure for density functional theory calculations, Comput. Mater. Sci. 50 (8) (2011) 2295–2310, https://doi.org/10.1016/j.commatsci.2011.02.023 http://www.sciencedirect.com/science/article/pii/S0927025611001133.

[15] J. Anubhav, O.S. Ping, C. Wei, M. Bharat, Q. Xiaohui, K. Michael, B. Miriam, P. Guido, R. Gian Marco, H. Geoffroy, G. Daniel, P.K. Kristin, Fireworks: a dynamic workflow system designed for high throughput applications, Concurr. Comput.: Practice Exp. 27 (17) (2015) 5037–5059, https://doi.org/10.1002/cpe.3505 https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3505.

[16] S.P. Ong, W.D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V.L. Chevrier, K.A. Persson, G. Ceder, Python materials genomics (pymatgen): A robust, open-source python library for materials analysis, Comput. Mater. Sci. 68 (2013) 314–319, https://doi.org/10.1016/j.commatsci.2012.10.028 http://www.sciencedirect.com/science/article/pii/S0927025612006295.

[17] K. Mathew, J.H. Montoya, A. Faghaninia, S. Dwarakanath, M. Aykol, H. Tang, I. heng Chu, T. Smidt, B. Bocklund, M. Horton, J. Dagdelen, B. Wood, Z.-K. Liu, J. Neaton, S.P. Ong, K. Persson, A. Jain, A high-level interface to generate, execute, and analyze computational materials science workflows, Comput. Mater. Sci. 139 (2017) 140–152, https://doi.org/10.1016/j.commatsci.2017.07.030 http://www.sciencedirect.com/science/article/pii/S0927025617303919.

[18] G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari, B. Kozinsky, AiiDA: automated interactive infrastructure and database for computational science, Comput. Mater. Sci. 111 (2016) 218–230, https://doi.org/10.1016/j.commatsci.2015.09.013 http://www.sciencedirect.com/science/article/pii/S0927025615005820.

[19] Materials studio, http://accelrys.com/products/collaborative-science/biovia-materials-studio/.

[20] MedeA, http://www.materialsdesign.com/medea.

[21] T. Mayeshiba, H. Wu, T. Angsten, A. Kaczmarowski, Z. Song, G. Jenness, W. Xie, D. Morgan, The MAterials Simulation Toolkit (MAST) for atomistic modeling of defects and diffusion, Comput. Mater. Sci. 126 (2017) 90–102, https://doi.org/10.1016/j.commatsci.2016.09.018 http://www.sciencedirect.com/science/article/pii/S0927025616304591.

[22] K. Mathew, A. Singh, J. Gabriel, K. Choudhary, S. Sinnott, A. Davydov, F. Tavazza, R. Hennig, Mpinterfaces: A materials project based python tool for high-throughput computational screening of interfacial systems, Comput. Mater. Sci. 122 (2016) 183–190, https://doi.org/10.1016/j.commatsci.2016.05.020.

[23] K. Choudhary, I. Kalish, R. Beams, F. Tavazza, High-throughput identification and characterization of two-dimensional materials using density functional theory, Sci. Rep. 7 (1) (2017) 5179, https://doi.org/10.1038/s41598-017-05402-0 http://europepmc.org/articles/PMC5507937.

[24] K. Choudhary, F.Y.P. Congo, T. Liang, C. Becker, R.G. Hennig, F. Tavazza, Evaluation and comparison of classical interatomic potentials through a user-friendly interactive web-interface, Sci. Data 4 (2017) 160125, https://doi.org/10.1038/sdata.2016.125.

[25] A. Goyal, P. Gorai, H. Peng, S. Lany, V. Stevanovi, A computational framework for automation of point defect calculations, Comput. Mater. Sci. 130 (2017) 1–9, https://doi.org/10.1016/j.commatsci.2016.12.040 http://www.sciencedirect.com/science/article/pii/S0927025617300010.

[26] E.B. Tadmor, R.S. Elliott, J.P. Sethna, R.E. Miller, C.A. Becker, The potential of atomistic simulations and the knowledgebase of interatomic models, JOM 63 (7) (2011) 17, https://doi.org/10.1007/s11837-011-0102-6.

[27] L.M. Hale, Z.T. Trautt, C.A. Becker, Evaluating variability with atomistic simulations: the effect of potential and calculation methodology on the modeling of lattice and elastic constants, Modell. Simul. Mater. Sci. Eng. 26 (5) (2018) 055003, http://stacks.iop.org/0965-0393/26/i=5/a=055003.

[28] F.D. Murnaghan, The compressibility of media under extreme pressures, Proc. Nat. Acad. Sci. 30 (9) (1944) 244–247, http://www.pnas.org/content/30/9/244.

[29] PEP 8 – style guide for python code, https://www.python.org/dev/peps/pep-0008/.

[30] F. Perez, B.E. Granger, IPython: A system for interactive scientific computing, Comput. Sci. Eng. 9 (3) (2007) 21–29, https://doi.org/10.1109/MCSE.2007.53.

[31] W. McKinney, Data structures for statistical computing in python, in: S. van der Walt, J. Millman (Eds.), Proceedings of the 9th Python in Science Conference, 2010, pp. 51–56.

[32] H. Nguyen, D.A. Case, A.S. Rose, NGLview interactive molecular graphics for jupyter notebooks, Bioinformatics 34 (7) (2018) 1241–1242, https://doi.org/10.1093/bioinformatics/btx789 arXiv:/oup/backfile/content_public/journal/bioinformatics/34/7/10.1093_bioinformatics_btx789/2/btx789.pdf.

[33] T.H. Group, Hierarchical data format, version 5, http://www.hdfgroup.org/HDF5/ (1997-NNNN).

[34] S. van der Walt, S.C. Colbert, G. Varoquaux, The numpy array: A structure for efficient numerical computation, Comput. Sci. Eng. 13 (2) (2011) 22–30, https://doi.org/10.1109/MCSE.2011.37.

[35] P. de Buyl, P.H. Colberg, F. Höfling, H5MD: A structured, efficient, and portable file format for molecular data, Comput. Phys. Commun. 185 (6) (2014) 1546–1553, https://doi.org/10.1016/j.cpc.2014.01.018 http://www.sciencedirect.com/science/article/pii/S0010465514000447.

[36] M. Atkinson, R. Morrison, Orthogonally persistent object systems, VLDB J. 4 (3) (1995) 319–401, https://doi.org/10.1007/BF01231642.

[37] J.D. Hunter, Matplotlib: A 2D graphics environment, Comput. Sci. Eng. 9 (3) (2007) 90–95, https://doi.org/10.1109/MCSE.2007.55.

[38] A. Glensk, B. Grabowski, T. Hickel, J. Neugebauer, Understanding anharmonicity in fcc materials: From its origin to ab initio strategies beyond the quasiharmonic approximation, Phys. Rev. Lett. 114 (2015) 195901, https://doi.org/10.1103/PhysRevLett.114.195901 https://link.aps.org/doi/10.1103/PhysRevLett.114.195901.

[39] J.E. Angelo, N.R. Moody, M.I. Baskes, Trapping of hydrogen to lattice defects in nickel, Modell. Simul. Mater. Sci. Eng. 3 (3) (1995) 289, https://doi.org/10.1088/0965-0393/3/3/001 http://stacks.iop.org/0965-0393/3/i=3/a=001.

[40] pyiron, http://pyiron.org.

[41] D.E. Smirnova, A.Y. Kuksin, S.V. Starikov, V.V. Stegailov, Z. Insepov, J. Rest, A.M. Yacout, A ternary EAM interatomic potential for UMo alloys with xenon, Modell. Simul. Mater. Sci. Eng. 21 (3) (2013) 035011, http://stacks.iop.org/0965-0393/21/i = 3/a = 035011.

[42] J.P. Perdew, K. Burke, M. Ernzerhof, Generalized gradient approximation made simple, Phys. Rev. Lett. 77 (1996) 3865–3868, https://doi.org/10.1103/PhysRevLett.77.3865 https://link.aps.org/doi/10.1103/PhysRevLett.77.3865.

[43] Y. Lysogorskiy, J. Janssen, T. Hammerschmidt, J. Neugebauer, R. Drautz, Transferability correlations of effective interaction models for Mo and Si (submitted for publication).

[44] Jupyterhub, https://github.com/jupyterhub/jupyterhub.