Nexus: a modular workflow management system for quantum simulation codes

Jaron T. Krogel

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Nexus: a modular workflow management system for quantum simulation codes

Jaron T. Krogel[a,*]

[a]*Materials Science and Technology Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA.*

Abstract

The management of simulation workflows represents a significant task for the individual computational researcher. Automation of the required tasks involved in simulation work can decrease the overall time to solution and reduce sources of human error. A new simulation workflow management system, Nexus, is presented to address these issues. Nexus is capable of automated job management on workstations and resources at several major supercomputing centers. Its modular design allows many quantum simulation codes to be supported within the same framework. Current support includes quantum Monte Carlo calculations with QMCPACK, density functional theory calculations with Quantum Espresso or VASP, and quantum chemical calculations with GAMESS. Users can compose workflows through a transparent, text-based interface, resembling the input file of a typical simulation code. A usage example is provided to illustrate the process.

Keywords: workflow management; high-throughput; productivity; automation; quantum simulation; electronic structure; quantum Monte Carlo; density functional theory; QMCPACK; Quantum Espresso; GAMESS; VASP

PROGRAM SUMMARY

Manuscript Title: Nexus: a modular workflow management system for quantum simulation codes
Authors: Jaron T. Krogel
Program Title: Nexus

*Corresponding author.
E-mail address: krogeljt@ornl.gov

Journal Reference:
Catalogue identifier:
Licensing provisions: University of Illinois/NCSA Open Source License
Programming language: Python
Computer: Workstations, clusters, and supercomputing resources at OLCF, ALCF, NERSC, NCSA, TACC, and RPI.
Operating system: Linux
RAM: 100 megabytes
Keywords: workflow automation, high-throughput, productivity, quantum simulation, electronic structure, quantum monte carlo, density functional theory, QMCPACK, Quantum Espresso, GAMESS, VASP
Classification: 6.5, 7.3
External routines/libraries: NumPy. Optionally SciPy, Matplotlib, and H5Py.
Nature of problem:
Quantum simulation workflows with multiple dependencies between various electronic structure codes. Job management of simulations on arbitrary workstation and supercomputing environments. Aggregation and analysis of output data.
Solution method:
Modular and extensible class-based framework to represent atomic structures, simulation instances, input files, output data, job requests, and host machines. Automatic identification of time ordered simulation networks (workflows) and simultaneous management of independent workflows including writing of input files, submission of jobs on available resources, job monitoring, and collection and post processing of output data. High-level and minimal user interface to generate input files, compose workflows, and perform data analysis.

1. Introduction

The management of scientific simulation workflows can be a labor intensive enterprise for the individual researcher. Many computational studies involve running disparate simulation codes, transferring outputted information into new simulation inputs, monitoring simulation jobs, and collecting data for analysis. Further, studies may involve many of these workflows, representing a significant management burden for the researcher. The automation of such processes can offer significant gains in productivity and allow researchers to focus more on problem design and interpretation of results than workflow execution.

Many workflow management systems already exist for general scientific workflows, *e.g.* for processing the massive amounts of data generated by

large scale experiments (see Refs. [1, 2] and references therein). Recently, there is a wave of interest in high-throughput style simulation in the electronic structure and quantum chemistry communities. This has resulted in a variety of simulation workflow software, sometimes tailored tightly to the desired application. Representative efforts include the Electronic Structure Project [3], the Harvard Clean Energy Project [4], AFLOW [5], the Quantum Materials Informatics Project [6], the Materials Project [7], and the Molecular Simulation Grid [8]. Researchers benefit from a range of tools that both improve productivity of general simulation work and provide access to high-throughput style calculations in a familiar framework. The present paper presents a new scientific workflow management system, called Nexus, which aims to be accessible to practitioners of computational physics who routinely work with major quantum simulation codes through text based input files or custom made driving scripts.

Nexus was designed in the context of quantum Monte Carlo [9] (QMC) calculations. Production QMC calculations often involve narrow and deep workflows of many chained simulations using disparate codes. The resulting design renders Nexus equally applicable to the wide and shallow workflows involved in high-throughput simulations. Nexus is capable of automated job management on workstations (by acting as a virtual queue), institutional clusters, and high-performance supercomputers at OLCF, ALCF, and NERSC, among others. Job management capabilities include handling dependencies between calculations, bundling compute jobs, and extracting results from completed calculations for analysis.

Users interact with Nexus by writing a Python [10] script which typically resembles a simple input file. More complex simulation workflows involving many dependencies can be readily created with standard if/else logic and for loops. Although Nexus and its user interface are written in Python, a detailed knowledge of the language is not required to use the system.

The modular, class-based design of Nexus permits many simulation codes to be driven within the same overall framework. Codes currently supported include QMCPACK [11] (wavefunction optimization [12], variational Monte Carlo [13], diffusion Monte Carlo [14, 15]), SQD (numerical Hartree-Fock [16, 17] for atoms, packaged with QMCPACK), and Quantum Espresso/PWSCF [18] (total energy or structural relaxation calculations via density functional theory [19, 20] (DFT)). Support for GAMESS [21, 22], VASP [23, 24, 25, 26], and the OPIUM [27] pseudopotential generator is at an early stage. The basic design, functionality, and user interface of Nexus are covered in more detail

in the sections that follow.

2. Program design

Nexus is an object-oriented Python code designed in a modular fashion to logically segregate important data and functions. Because of its dynamic class-based design, Nexus is highly extensible, enabling support for new machines and simulation codes in a straightforward way. Python was selected as the development language because it enables compact and transparent coding techniques that would amount to metaprogramming in other languages such as C++. Primarily because the language is dynamic, it is also possible to create a uniform and general programming environment by using a high-level base class to encapsulate many common functions, such as hierarchical printing, saving, and loading of data. This improves both the development and user environments. For example, all Nexus objects are interactively browsable much like navigating and querying a traditional file system.

The class structure of Nexus is summarized in table 1. The purpose of each class is provided in the second column. Detailed descriptions of these classes are contained in the subsections below. The dynamic interaction between these elements during Nexus runtime execution is the subject of section 3.

Class	Purpose
Structure	Represents atomic structure. Permits structural manipulations.
PhysicalSystem	Represents whole system: electrons + atoms. Provides species information.
SimulationInput	Represents input file for a given simulation code. Supports file read, modification, and write. Base class.
SimulationAnalyzer	Represents output data of a given simulation code. Used for data analysis. Base class.
Simulation	Represents a single simulation run. Tracks simulation workflow dependencies and supports output monitoring. Base class.
Job	Represents a generic machine job. Communicates requested resources (nodes, walltime, etc.) and tracks job state.
Machine	Represents a generic machine. Submits jobs and monitors job queue. Base class.
Workstation	Represents a workstation. Maintains virtual job queue.
Supercomputer	Represents a generic supercomputer. Extended to support local environment. Base class.
ProjectManager	Manages total simulation workflow process. Retains memory of workflow state.

Table 1: Major classes of the Nexus program.

The relationship between Nexus classes is demonstrated more clearly through a Unified Modeling Language (UML) class diagram, as shown in figure 1. For those unfamiliar with UML, classes connected by solid lines are related by composition. Open diamonds refer to weak composition (e.g. via pointers) and solid diamonds refer to full containment. The label above each line is the name of the contained data member and the text below the line refers to its multiplicity. For example, a ProjectManager instance has a data member called `cascades` that is effectively a list of pointers to Simulation instances. The "*" used here indicates that there can be zero, one, or many such Simulation instances. By contrast, a PhysicalSystem instance directly contains a single Structure instance called `structure`. Dashed arrows refer to weak association between classes. For example, a PhysicalSystem instance is not contained by any SimulationInput instances, but is rather passed in as

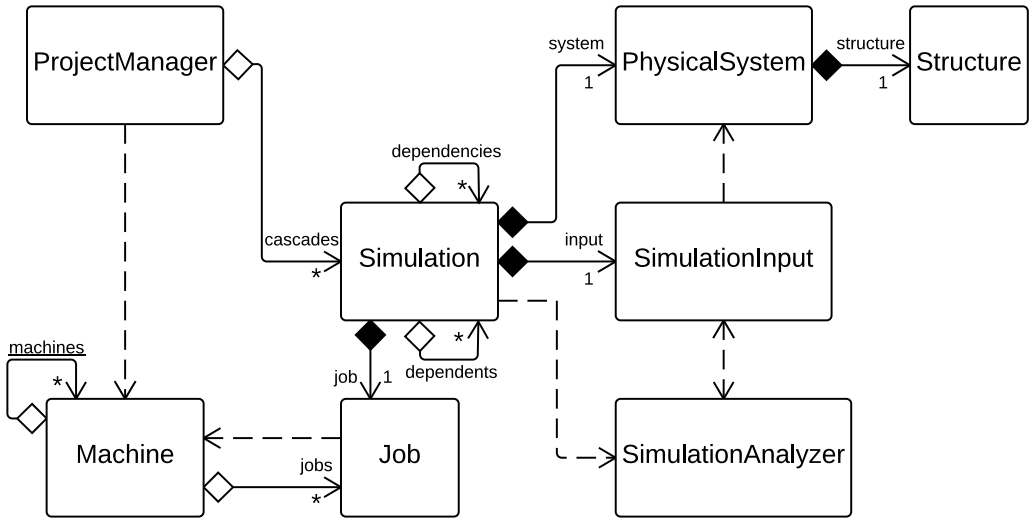

Figure 1: UML class diagram for select Nexus classes.

an argument to one of SimulationInput's member functions.

2.1. Structure class

The Structure class represents the atomic structure of a crystal or molecule. It contains a list of atomic types and positions, a matrix representing the simulation cell (if any), and a set of kpoints and corresponding weights if the system is periodic. It also tracks whether atoms are free to move and in which directions (useful for molecular dynamics or structural relaxation). Because the global concept of a periodic crystal structure can have many equivalent representations (*e.g.* primitive cell, conventional cell, supercell, etc.), a given Structure object optionally maintains a hierarchy of nested equivalent representations (folded structures). Each equivalent structure contains a set of appropriately mapped kpoints for the cell. This is particularly useful for QMC calculations where the electronic orbitals are generated in DFT within a primitive cell to save memory but the actual QMC calculation is performed explicitly in an equivalent supercell. Several operations are made available through the Structure class to manipulate atomic structure, create supercells, add k-points to a cell, or get detailed atomic information (*e.g.* distance tables). An abbreviated summary of available functionality is listed in Appendix A.

2.2. PhysicalSystem class

This class represents the totality of the physical system under study. A PhysicalSystem object contains a Structure object to provide atomic structure information. Electronic structure information is currently represented in a coarse way; only the total charge and spin polarization are recorded. It also tracks information about each species of particle such as the mass, charge, or effective valence charge for an ion if pseudopotentials are used. Often many simulations of different types are performed on the same physical system. This class provides a uniform description of the physical system regardless of the target simulations so the user only has to specify this information once. Its main task in the workflow environment is to interact with simulation input objects to fill out much of the structural and electronic information of an input file.

2.3. SimulationInput class

The SimulationInput class represents the input file of a simulation code. Derived classes are created to represent the input files of specific codes, such as QMCPACK or Quantum Espresso. The design of a derived SimulationInput class typically includes a general base class that represents a generic subsection of input, followed by a series of derived classes for specific sections. These subsection derived classes often contain little more than the direct annotation of input keywords and resemble a reference specification for the input file as a whole. The internal representation of a SimulationInput instance is a nested collection of custom Python container objects. A minimal representation is sought wherever possible. In practice the nested objects only contain keyword-value pairs, where the value is typically a basic type (`int`, `float`, `bool`, `str`), an array of basic types, or another container object if the input file is hierarchically structured.

Each derived class of SimulationInput aims to be able to read an arbitrary input file from disk into the object representation and write a general input file to disk from the object form. An input file object is intended for interactive use and is browsable at the command line. SimulationInput objects are also dynamic, so any keyword-value pair can be directly added to an existing object in a simple way. A faithful object representation of an input file has the advantage of native, high-level, scripted search and modification of an input file in numerical or other appropriate data types prior to writing, avoiding text-based manipulation entirely. This makes it easy to add or remove whole input file sections or transfer them between files. This object

representation also enables the direct generation of arbitrary input files in the abstract. Input generator functions are available for each major code supported by Nexus. The generator functions can accept an arbitrary number of inputs which are screened against the native keywords of the input file and other variables internal to Nexus. This has the advantage of providing a minimal, yet general interface to users that can be augmented by expert knowledge in the background (*e.g.* context dependent defaults guided by expert judgment). SimulationInput objects also facilitate deeper automated introspection of input files to determine the type of simulation being performed and its expected outputs, which is useful for workflow management.

2.4. SimulationAnalyzer class

The SimulationAnalyzer class represents the output data of a generic simulation code. It permits access to output in terms of native scalar and array numerical data types. The object substructure of a SimulationAnalyzer instance generally mirrors structure of output data. An analyzer object typically performs a custom coded read of the main log file of a simulation run and extracts numerical data. It can also supply direct access to machine-encoded data formats such as the XML [28] output of Quantum Espresso and VASP or the HDF5 [29] output of QMCPACK. Wherever possible a SimulationAnalyzer object performs analysis of the data and stores a representation of the results in anticipation of later user access. Upon completion of a simulation run, its corresponding analyzer object is saved to disk in the standard Python binary (pickle) format. A SimulationAnalyzer class may also supply plotting functions tailored to the application.

2.5. Simulation class

Simulation objects represent individual simulations, planned or in execution. Figure 2 shows the major Simulation classes. Simulation objects are arranged together in time-ordered networks or cascades where each simulation may depend on the result of another (circular dependencies are not allowed). In the context of Nexus, a "cascade" is the data representation of a scientific workflow that will be carried out on a particular machine. An example of a simulation workflow, or cascade, is shown in figure 3. Simulation objects are linked to other Simulation objects that are "dependencies" or "dependents" (refer to figure 1 for these member variables). In the case of figure 3, the Pwscf Simulation object for an NSCF calculation has two dependencies (a Pwscf simulation generating a relaxed structure and another

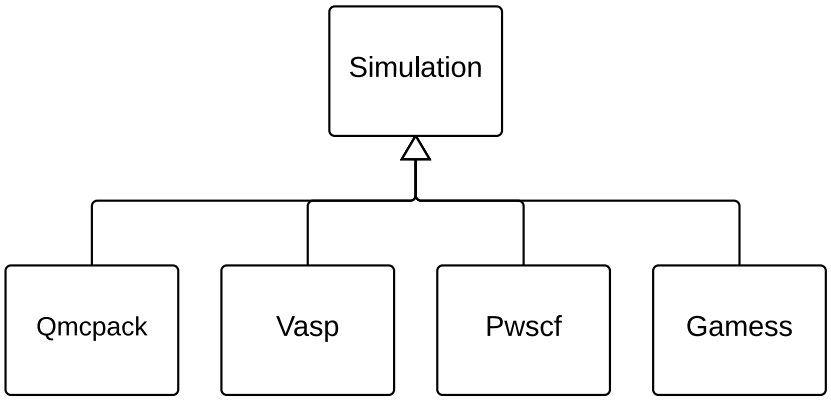

Figure 2: Inheritance diagram for major Simulation classes.

Pwscf simulation generating a charge density) and two dependents (a Qmcpack optimization run and a Qmcpack DMC run).

The Simulation base class fills many roles during the workflow execution process. It is responsible for writing the input file to disk via a contained SimulationInput object, copying in other needed files (such as pseudopotentials) prior to execution, supplying job information to the host Machine object via a contained Job object, and collecting data for analysis via a contained SimulationAnalyzer object. Additional functions implemented in derived Simulation objects include judging the success or failure of a completed simulation run and handling the processing of dependency information between runs. A given Simulation object incorporates products of other completed simulation runs into its input file (*e.g.* a relaxed structure) and packages information about its own products (*e.g.* a converged charge density) for use by subsequent simulations. The state of a simulation is regularly saved to disk during workflow progression which allows Nexus to resume interrupted workflows without rerunning completed simulation runs.

At the beginning of this section, figure 1 gives a graphical representation of the central position of the Simulation class in the operations of Nexus. For a listing of all simulation codes and supporting tools that can be managed

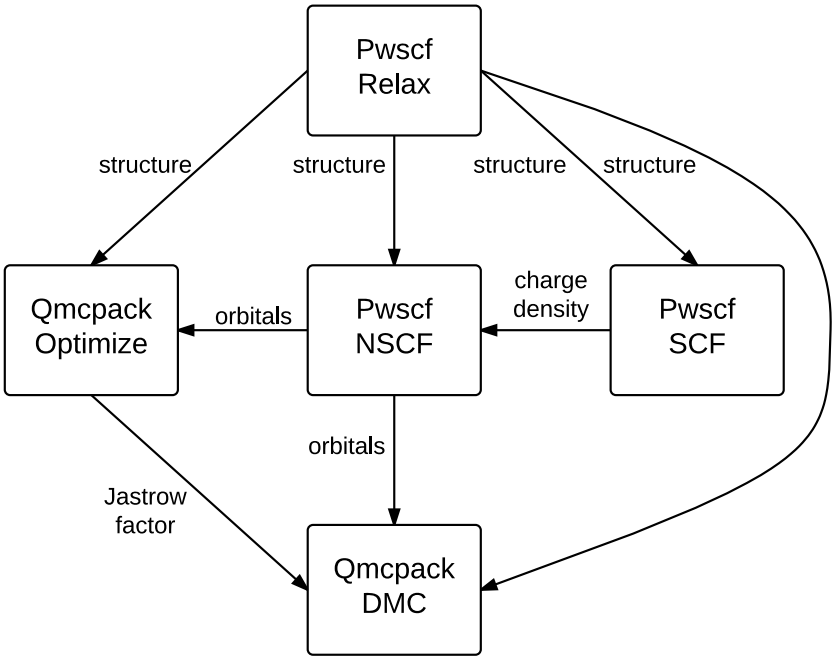

Figure 3: An example Nexus workflow/cascade involving QMCPACK and PWSCF. The arrows and labels denote the flow of information between the simulation runs.

with Nexus, refer to section 4.

2.6. Job class

The Job class represents the computing resources required to run a simulation and the current state of a running process from the perspective of the host machine. It contains the desired number of nodes/cores and OpenMP threads, the allotted time, and the name of the batch queue, if any. It also contains user defined flags to pass to the application, options to `mpirun`/`aprun` or similar, options to `qsub` or similar, and any environment variables for the job. The job state (submitted, finished, etc.) is updated

by the host Machine object. Interactions between the host Machine object and each Simulation object, such as job submission and notification of job completion, is mediated by a corresponding Job object.

2.7. Machine class

The machine class represents the host machine simulations are executed on. A Machine object is responsible for submitting and monitoring jobs on the physical host machine. Each Machine object maintains a virtual queue of internally submitted jobs (a list of Job objects) which mirrors the state of execution on the physical machine. Two important Machine subclasses corresponding to workstations and supercomputers (batch machines) are covered below.

2.8. Workstation class

A Workstation object represents a desktop machine with a given number of cores. It spawns multiple processes/jobs until all cores of the physical machine are occupied. Whenever a process completes, the Workstation object initiates new processes to fill the gap. In this way the Workstation class acts as a scheduler for desktop machines or small clusters. Individual jobs are submitted on the physical machine with calls to `mpirun`.

The majority of Nexus users will interact with the Workstation class, which provides a uniform experience across machines, including locally managed clusters. For those working in specialized high performance computing environments, derived Supercomputer classes offer interfaces tailored to each local environment.

2.9. Supercomputer class

This class represents a supercomputer composed of many nodes. Jobs are submitted from the virtual queue to the actual queue of the machine by writing submission scripts (*e.g.* for PBS [30] batch queues), and submitting the jobs to the scheduler (*e.g.* via `qsub`). Job progress is monitored by occasional queries of the actual queue (*e.g.* via `qstat`). In cases where a limited number of jobs are permitted to exist in the actual queue, remaining jobs are retained in the internal virtual queue until slots become available in the actual queue. Upon job submission, the queue id is captured and stored so that job monitoring and workflow progression can resume unaffected if the Nexus host process is interrupted. Derived Supercomputer classes allow for variations among supercomputing environments while exposing a

common interface to the user. To the extent possible, commonly encountered environment functionality, such as the use of the `qstat` queue querying tool, is handled by the Supercomputer base class. Derived classes are usually quite lightweight, often just containing details about the local batch submission scripts. See section 4 for a list of the currently supported supercomputing environments.

A word about the relationship between the Simulation, Job, and Supercomputer classes is due here. Ensuring that any of N simulation codes can be executed in each of M supercomputing environments naively requires run instructions for $N \times M$ simulation/environment pairs to be coded explicitly in Nexus. This problem is simplified through the polymorphism of the Simulation and Supercomputer classes. Upon instantiation, a derived Simulation object (*e.g.* Pwscf, Qmcpack, etc.) populates its contained Job object with information regarding the launch of its application binary (*e.g.* `pw.x`, `qmcapp`, etc.). This information includes environment variables and command line arguments required by the application. Just prior to job submission to the scheduler, a derived Supercomputer object representing the local machine extracts information from the Job object to populate the submission script's header (# nodes, walltime, queue, etc.), setup environment variables, and combine the applications command line inputs with the application launcher local to the machine (such as `aprun` or `mpirun`). In this way, Nexus only requires the separate implementation of N Simulation classes and M Supercomputer classes.

2.10. ProjectManager class

The ProjectManager class actively manages simulation workflows. It contains a list of all Simulation objects and partitions them into independent cascades. Through repeated time ordered traversals of the cascades, the project manager progresses each Simulation object to run completion and orchestrates the communication of dependency information between simulations. The ProjectManager also closely interacts with the host Machine object to identify completed cascades/workflows. The following section goes into more detail regarding active workflow management and the dynamic interaction between the ProjectManager and the other Nexus classes.

3. Runtime behavior

The active management stage of Nexus execution consists of a balance of tasks between the ProjectManager and multiple Machine, Job, and Simulation objects. The active management stage is initiated by a call to the `run_project` function within a user's input script (see sections 5 and 6 for further details). A UML activity diagram of the `run_project` function can be found in figure 4. There, the flow of control is partitioned according to the class performing each activity/function.

3.1. Cascade initialization and polling framework

When control is handed to Nexus, the ProjectManager identifies independent cascades (disconnected workflow graphs) and the "sources" of each cascade. A cascade source is a simulation that depends on no others (note that cascades can have multiple sources). Following identification, the simulation cascades are initially traversed several times to gather information and ensure that there will be no conflicts. Cascade traversal is performed in such a way that a simulation object is visited only after all simulations it is dependent on have been visited. During the first traversal, the ProjectManager checks that simulations performed in the same directory will not attempt to create files with the same names to avoid data loss and other potential side effects. Next if the user has marked any simulations as "blocked" (*i.e.* requested that a simulation not be executed), all dependents of blocked simulations down the cascade will also be blocked. Then the ProjectManager loads the prior state image of each simulation from disk, if present, so each simulation cascade can progress from where it left off. Finally, the dependencies comprising the connections between the simulations in the cascade/workflow are checked. Here it is confirmed that the dependency requests being made by each simulation (*e.g.* for a relaxed structure, a converged charge density, or an optimized Jastrow factor) correspond to items actually being produced by earlier simulations. Once all checks have been passed, active job submission and monitoring begin.

The simulation workflows are managed by Nexus within a polling framework. At a user provided polling frequency, the Nexus host process wakes from sleep and checks the job queue on the host machine. The internal job queue in the Machine object is updated to reflect any changes in job status ("query queue" in figure 4). The act of checking the host job queue currently consists of querying running processes on a workstation, or parsing

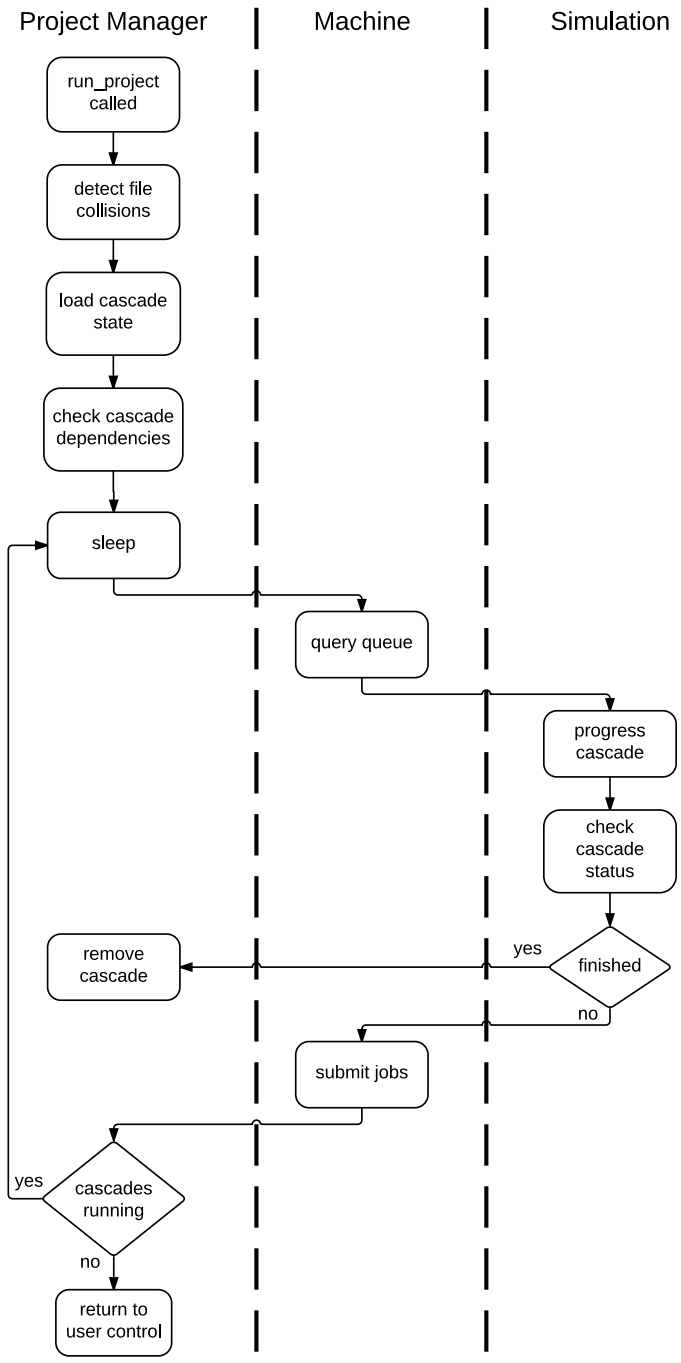

Figure 4: UML activity diagram for the `run_project` function where active workflow/cascade management occurs through a polling cycle. More detail regarding the "progress cascade" activity/function is given in figure 5.

the text output of command line tools such as `qstat` in supercomputing environments.

The next task performed during the poll is to take each active simulation in each cascade through a progression of operations ("progress cascade" in figure 4), such as writing input files, submitting a job to the virtual Machine queue, and collecting output data. This fine grained process, which proceeds incrementally at each successive poll, is described in more detail in the next section (also see figure 5). If all simulations downstream a cascade source have completed ("check cascade status" in figure 4), that cascade is removed from the list of "progressing" or actively managed cascades within the ProjectManager ("remove cascade" in figure 4).

At the end of each poll, any jobs waiting in the virtual Machine queue are either launched directly (Workstation), or submitted to the actual queue (Supercomputer) of the physical machine ("submit jobs" in figure 4). The ProjectManager exits the polling cycle when all cascades have completed. At this point control is returned to the user script, where further post processing of output data can be performed.

Overall the demands of the Nexus host process on local computing resources are very light. In practice the load at each poll amounts to a small fraction (<10%) of processing resources on a single core for a short amount of time (<5 seconds). If extremely low overhead is required, the time between polls can be increased to several minutes.

3.2. Simulation action stages

During the "progress cascade" action performed at each poll, all simulation cascades are partially traversed and uncompleted actions are performed for appropriate simulations. Each Simulation object is eventually advanced through seven successive action stages: processing information from its Simulation dependencies to inform the current run, writing input files, copying in other needed files (such as pseudopotentials), job submission, collecting lightweight output files, and analyzing the outputted data. The UML activity diagram corresponding to this process is given in figure 5. A given Simulation object begins its progression through these stages only when all other simulations it depends on have completed.

During the first poll required results (such as relaxed structures, etc.) are collected from earlier completed runs and combined with user input to write input files. Needed files are copied to the local directory and the Simulation's

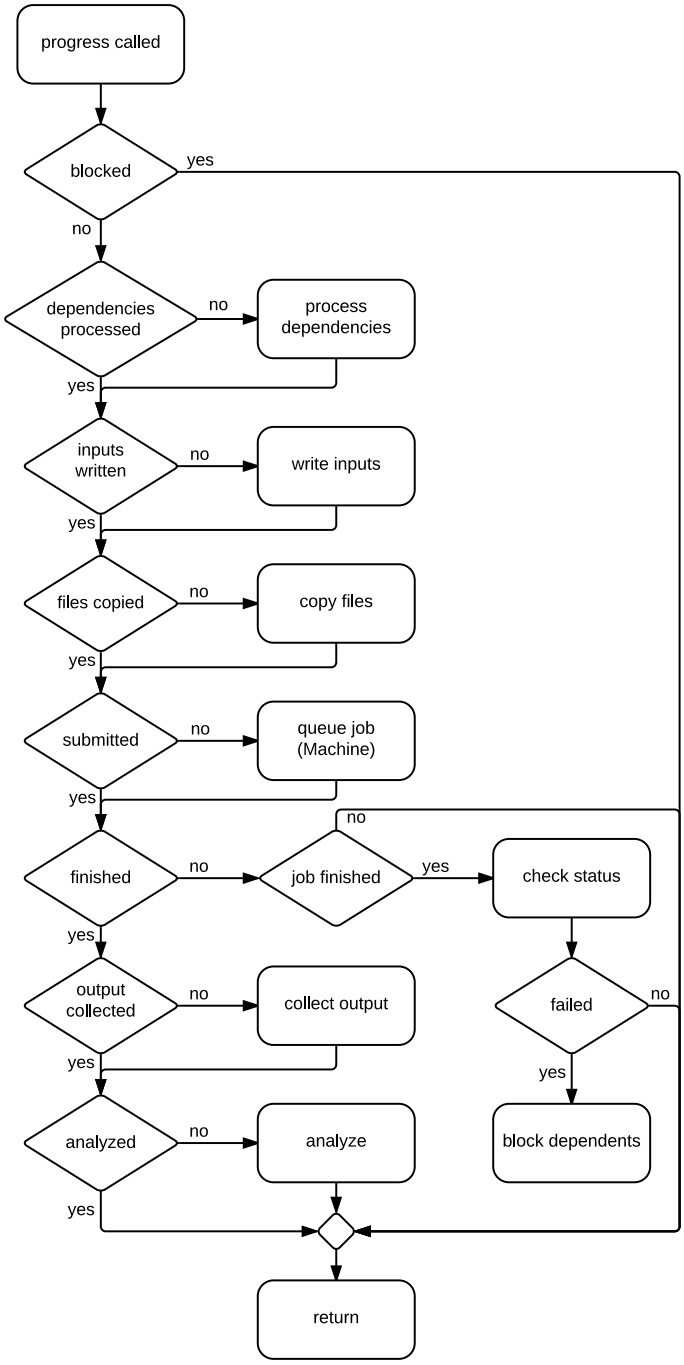

Figure 5: UML activity diagram for the "progress cascade" action from figure 4. The activity shown is for a single Simulation object during a given poll.

Job object is submitted to the virtual queue of the appropriate Machine object.

On subsequent polls, a simulation in the submission phase is monitored for successful completion. Often this involves checking for the presence of expected output files or characteristic signs of completion in the output log ("finished" check in figure 5) in addition to checking whether the process in the batch/system queue has ceased execution ("job finished" check in figure 5). In the event of unsuccessful execution all downstream (dependent) simulations from a failed run are blocked from running.

Following successful execution, the lighter weight data files are optionally copied to a results directory for transfer or backup by the user. At this point the simulation output is also analyzed and condensed into a set of numerical results that are saved on disk in a structured data format. This "analyzed" output is used to furnish computed quantities as inputs to subsequent simulations and also provides the user with ready access to numerical data for further processing. At each action stage an image of simulation status is written to disk to record new progress.

4. Current capabilities

Nexus is currently capable of driving a handful of simulation codes on workstations, clusters, and several supercomputing environments. Most users should be able to productively use Nexus on their local workstations or group clusters without modification. The usage example provided in section 6 can be straightforwardly expanded to more realistic use cases or core counts of a local cluster. Note that regardless of the environment, the Nexus host process is executed locally on the machine or within an interactive batch job.

Since many high performance machines (supercomputers) are independently maintained, the user environment can be site-specific. For this reason, a separate Supercomputer class has been created for each machine. Most currently supported machine environments are shown in table 2. This list will be expanded as new resources become available.

If a particular HPC machine/center of interest is not yet supported, the interested user can follow the pattern set out for already supported systems at the end of the `machines.py` file in the Nexus source distribution to include the new machine. Adding support for a new environment can often be accomplished with a small amount of code (≈30 lines) that maps data from a `Job` object to the appropriate fields of the local job submission file. Users

are also welcome to contact the author directly for assistance if a new machine interface is desired. With sufficient input, more generic Supercomputer classes will be created in the future to handle broad ranges of machines that fully share environment characteristics.

Location	**Machines**
anywhere	Linux workstation/cluster
Oak Ridge Leadership Computing Facility	Titan, Eos
Argonne Leadership Computing Facility	Mira, Vesta, Cetus
National Energy Research Scientific Computing Center	Hopper, Edison
National Center for Supercomputing Applications	Blue Waters, Taub
Texas Advanced Computing Center	Longhorn
Rensselaer Polytechnic Institute	AMOS

Table 2: Computing environments currently supported by Nexus.

Table 3 contains a listing of the simulation codes that have a detailed Nexus interface. The types of runs typically performed are also noted. These include wavefunction optimization (Opt.), variational Monte Carlo (VMC), diffusion Monte Carlo (DMC), Hartree Fock (HF), self-consistent density functional theory (SCF,DFT), non-self-consistent DFT (non-SCF), DFT structural relaxation (relax), nudged elastic band (NEB), and multi-configurational self-consistent field (MCSCF). The run types listed represent a subset of the types of calculations that can actually be driven by Nexus. The interface to each code's input file is actually quite general, so essentially all types of runs supported by the code can be driven with Nexus. The last two columns of table 3 list the types of information that can be communicated between different simulations (dependencies). These include relaxed atomic structures (structure), single particle orbitals or Bloch states (orbitals), electronic charge densities (density), and optimized Jastrow correlation factors (Jastrow). The first column lists all results that can be produced by a given simulation code and the second lists the types of products it accepts as input (consumes). Complex simulation workflows/cascades can be constructed by making "consumers" dependent upon "producers" for appropriate simulation results (products).

Simulation Code	Supporting Tools	Run Types	Produces	Accepts
QMCPACK	pw2qmcpack wfconvert convert4qmc	Opt. VMC DMC	Jastrow	structure orbitals Jastrow
SQD		atomic HF	orbitals	
Quantum Espresso (PWSCF)		relax SCF non-SCF	structure density orbitals	structure density
GAMESS		HF, DFT MCSCF	orbitals	orbitals
VASP		relax SCF NEB	structure	structure

Table 3: Simulation codes and related tools currently supported by Nexus. Calculation (run) types as well as simulation products and accepted workflow dependencies are listed. Quantities listed under "Produces" and "Accepts" correspond to some, but not all, run types.

5. User interface

Nexus presents a simple interface to the user. Users interact with Nexus by writing a Python script that often resembles an input file. A sample script is shown in Example 1. In the example, Quantum Espresso is used to first relax the atomic structure of a small diamond vacancy cell, followed by a total energy calculation with a larger plane-wave cutoff. User's can have Nexus create, manage, and complete simulation workflows similar to those in Example 1 by running the script (*i.e.* by invoking "`./diamond.py`" or "`python diamond.py`" at the command line) following installation (see section 6 for installation details).

```
#! /usr/bin/env python

from nexus import *

settings(
    pseudo_dir    = './pseudopotentials',
    status_only   = 0,
    generate_only = 0,
    sleep         = 3,
    machine       = 'ws16'
```

```
    )

relax_job = Job(cores=16,app='pw.x')
scf_job   = Job(cores=16,app='pw.x')

dia16 = generate_physical_system(
    structure = './d16vac.POSCAR',
    C         = 4
    )

relax = generate_pwscf(
    identifier   = 'relax',
    path         = 'diamond_vacancy/relax',
    job          = relax_job,
    input_type   = 'generic',
    calculation  = 'relax',
    ion_dynamics = 'bfgs',
    input_dft    = 'lda',
    ecutwfc      = 35,
    conv_thr     = 1e-6,
    system       = dia16,
    pseudos      = ['C.BFD.upf'],
    kgrid        = (2,2,2),
    kshift       = (0,0,0),
    )

scf = generate_pwscf(
    identifier   = 'scf',
    path         = 'diamond_vacancy/scf',
    job          = scf_job,
    input_type   = 'generic',
    calculation  = 'scf',
    input_dft    = 'lda',
    ecutwfc      = 75,
    conv_thr     = 1e-7,
    system       = dia16,
    pseudos      = ['C.BFD.upf'],
    kgrid        = (2,2,2),
    kshift       = (0,0,0),
    dependencies = (relax,'structure')
    )

run_project(relax,scf)
```

Example 1: Nexus input script for structural relaxation of a diamond vacancy (16 atom cell) with PWSCF followed by self-consistent DFT using a higher plane-wave energy cutoff (`diamond_vacancy.py`).

Each script begins with imports from the main Nexus module. Items imported include the interface to provide settings to Nexus, helper functions to make objects representing atomic structures or simulations of particular types (*e.g.* QMCPACK or VASP), and the interface to provide simulation workflows to Nexus for active management. Elements can be imported separately by name or, as in Example 1, an import of all names can be accomplished with the abbreviated "`from nexus import *`". The user provides a few important pieces of information to Nexus through the `settings` function, such as the identity of the local machine, the directory where pseudopotentials for the current project are located (if any), and a few flags that control the behavior of the ProjectManager. These flags are covered in more detail in the usage example in section 6.

Following the settings, the user defines the atomic structure to be studied (whether generated or read in) which can be used to form input to various simulations (*e.g.* DFT and QMC) performed on the same system. Next simulation inputs are specified. In addition to application specific keywords, some information is always provided, such as the path where the simulation will be performed, computational resources required by the simulation job, and an identifier to differentiate between simulations (must be unique only for simulations occurring in the same directory). At this point, the user can also specify dependencies between jobs, if any. In Example 1, the final self-consistent DFT calculation depends on the relaxed atomic structure produced in the prior optimization run.

More complicated workflows or scans over parameters of interest can be created with for loops and if-else logic constructs. It should be stressed that no simulation jobs will executed upon creation of the simulation objects `relax` and `scf`. These objects only contain a request that a simulation be performed and provide sufficient information to accomplish that task. Simulation jobs are actually executed when the corresponding simulation objects are passed to the `run_project` function. Within the `run_project` function, most of the operations unique to Nexus are actually performed, as outlined in section 3. If a Nexus script is interrupted while simulations are running, it can simply be reinvoked and monitoring of the jobs will resume. When the `run_project` function returns, all simulations should be complete. Following the call to `run_project`, the user can perform data analysis tasks, if desired, as the analyzer object associated with each simulation contains a collection of post-processed output data rendered in numeric form (ints, floats, numpy arrays) and stored in a structured format.

The modular structure of the input format makes it simple to switch between host machines, atomic structures, or simulation packages. Example 1 is setup to run on a workstation with 16 cores. If another machine is desired (and supported) one can give its name in settings and appropriately adjust the job requests:

```
settings(
    ...
    machine       = 'titan',
    account       = 'ACC001'
    )

relax_job = Job(nodes=1,hours=1,queue='batch',app='pw.x')
scf_job   = Job(nodes=1,hours=1,queue='batch',app='pw.x')
```

Running bulk diamond in both simulations instead of a vacancy requires a small change:

```
dia16 = generate_physical_system(
    structure = './d16bulk.POSCAR',
    C         = 4
    )
```

Obtaining a relaxed structure with VASP instead of PWSCF is also straightforward (as `C.POTCAR` is proprietary, access is not provided here):

```
relax_job = Job(cores=16,app='vasp')
...
relax = generate_vasp(
    identifier   = 'relax',
    path         = 'diamond_vacancy/relax',
    job          = relax_job,
    input_type   = 'generic',
    istart       = 0,
    icharg       = 2,
    encut        = 450,
    nsw          = 5,
    ibrion       = 2,
    isif         = 2,
    system       = dia16,
    pseudos      = ['C.POTCAR'],
    kcenter      = 'monkhorst',
    kgrid        = (2,2,2),
    kshift       = (0,0,0),
    )
```

This user level modularity is especially advantageous for large projects that are typically composed of more complicated workflows spanning multiple systems, simulation codes, and host machines. The section that follows provides a more "hands on" experience with the Nexus user interface.

6. Usage example

By following the instructions contained in this section the reader can execute a simple workflow with Nexus. Much more complicated workflows have been driven with Nexus in production calculations involving QMCPACK [31, 32, 33, 34, 35, 36], Quantum Espresso [31, 32, 34, 35, 36], and VASP [37]. The workflow presented here is intended to illustrate basic Nexus usage. The example workflow has three stages: (1) orbital generation in a primitive (2 atom) cell of diamond with PWSCF, (2) conversion of the orbitals from the native PWSCF format to the ESHDF format that QMCPACK reads, and (3) a minimal variational Monte Carlo (VMC) run of a 16 atom supercell of diamond with QMCPACK.

The Nexus input script corresponding to this workflow is shown in example 2. The script is similar to the one discussed in section 5, differing mainly in the name-by-name imports and the description of the physical system. Instead of reading an external VASP POSCAR file, the structure is specified in a format native to Nexus. This format includes the unit system (Angstrom in this case), the three vectors comprising the axes of the simulation cell, and the names and positions (Cartesian coordinates) of the atoms involved. The use of "`tiling=(2,2,2)`" communicates the request that a $2\times2\times2$ supercell be constructed out of the specified 2 atom primitive cell. The k-point grid applies to the supercell and is comprised of a single k-point. The eight corresponding primitive cell images of this k-point are determined automatically by Nexus. The text "`C = 4`" specifies the number of valence electrons for the carbon pseudopotential. The pseudopotential files used in this example (`C.BFD.*`) have been adapted from an open access pseudopotential database [38] for use in PWSCF and QMCPACK. Since the PhysicalSystem object, "`dia16`", contains both the supercell and its equivalent folded/primitive version, the PWSCF DFT calculation will be performed in the primitive cell to save memory for the subsequent VMC calculation of the full supercell performed with QMCPACK.

```
#! /usr/bin/env python

```

```
from nexus import settings,Job,run_project
from nexus import generate_physical_system
from nexus import generate_pwscf
from nexus import generate_pw2qmcpack
from nexus import generate_qmcpack,vmc

settings(
    pseudo_dir    = './pseudopotentials',
    status_only   = 0,
    generate_only = 0,
    sleep         = 3,
    machine       = 'ws16'
    )

dia16 = generate_physical_system(
    units  = 'A',
    axes   = [[ 1.785,  1.785,  0.   ],
              [ 0.   ,  1.785,  1.785],
              [ 1.785,  0.   ,  1.785]],
    elem   = ['C','C'],
    pos    = [[ 0.    ,  0.    ,  0.    ],
              [ 0.8925,  0.8925,  0.8925]],
    tiling = (2,2,2),
    kgrid  = (1,1,1),
    kshift = (0,0,0),
    C      = 4
    )

scf = generate_pwscf(
    identifier   = 'scf',
    path         = 'diamond/scf',
    job          = Job(cores=16,app='pw.x'),
    input_type   = 'generic',
    calculation  = 'scf',
    input_dft    = 'lda',
    ecutwfc      = 200,
    conv_thr     = 1e-8,
    nosym        = True,
    wf_collect   = True,
    system       = dia16,
    pseudos      = ['C.BFD.upf'],
    )

conv = generate_pw2qmcpack(
    identifier   = 'conv',
    path         = 'diamond/scf',
    job          = Job(cores=1,app='pw2qmcpack.x'),
    write_psir   = False,
    dependencies = (scf,'orbitals')
```

```
    )

qmc = generate_qmcpack(
    identifier   = 'vmc',
    path         = 'diamond/vmc',
    job          = Job(cores=16,threads=4,app='qmcapp'),
    input_type   = 'basic',
    system       = dia16,
    pseudos      = ['C.BFD.xml'],
    jastrows     = [],
    calculations = [
        vmc(
            walkers     =   1,
            warmupsteps =  20,
            blocks      = 200,
            steps       =  10,
            substeps    =   2,
            timestep    =  .4
            )
        ],
    dependencies = (conv,'orbitals')
    )

run_project(scf,conv,qmc)
```

Example 2: Nexus input script for the usage example (`diamond.py`). DFT is performed in diamond primitive cell followed by VMC in a 16 atom supercell.

To fully execute the usage example provided here, copies of PWSCF, QMCPACK, and the orbital converter pw2qmcpack will need to be installed on the local machine. The example assumes that the executables are in the user's `PATH` and are named `pw.x`, `qmcapp`, and `pw2qmcpack.x`. The required source files and installation instructions are provided at `http://qmcpack.org/` and `http://www.quantum-espresso.org/`. A test of Nexus including the generation of input files, but without actual job submission, can be performed without installing these codes. However, Python itself (Python 2.7+, `https://www.python.org/`) and NumPy (`http://www.numpy.org/`) are required to run Nexus. The example also assumes the local machine is a workstation with 16 available cores ("`ws16`"). If fewer than 16 cores are available, *e.g.* 4, change the example files to reflect this: `ws16`→`ws4`, `Job(cores=16,...)`→`Job(cores=4,...)`.

Following download, unpack `nexus.tgz` and install Nexus by adding "`/your/path/to/nexus/library`" to the `PYTHONPATH` environment variable. To check that Python can find Nexus, type "`python`" at the command line

followed by "`from nexus import *`". Nexus should be available for use if the import proceeds without incident. See the `README` for further details.

In this example, we will run Nexus in three different modes:

1. status mode: print the status of each simulation and then exit (`status_only=1`).
2. generate mode: generate input files but do not execute workflows (`generate_only=1`).
3. execute mode: execute workflows by submitting jobs and monitoring simulation progress (`status_only=0, generate_only=0`).

Only the last mode requires executables for PWSCF and QMCPACK.

First, run Nexus in status mode. Enter the `example` directory, open `diamond.py` with a text editor and set "`status_only`=1". Run the script by typing "`./diamond.py`" at the command line and inspect the output. The output should be similar to the text below:

```
  Pseudopotentials
    reading pp:  ./pseudopotentials/C.BFD.upf
    reading pp:  ./pseudopotentials/C.BFD.xml

Project starting
  checking for file collisions
  loading cascade images
    cascade 0 checking in
  checking cascade dependencies
    all simulation dependencies satisfied
  cascade status
    setup, sent_files, submitted, finished, got_output, analyzed
    000000  scf  ./runs/diamond/scf
    000000  conv  ./runs/diamond/scf
    000000  vmc  ./runs/diamond/vmc
    setup, sent_files, submitted, finished, got_output, analyzed
```

The binary string "000000" indicates that none of the six stages of simulation progression have been completed. These stages ("`setup`", "`sent_files`", "`submitted`", "`finished`", "`got_output`", and "`analyzed`") have been explained in section 3. In a production setting, this mode is useful for checking the status of current workflows/cascades prior to adding new ones. It is also useful in general for detecting any problems with the Nexus input script itself.

Next, run the example in generate mode. Set "`status_only`=0" and "`generate_only`=1", then run the example script again. Instead of showing workflow status, Nexus will now perform a dry run of the workflows by generating all of the run directories and input files. The output should contain text similar to what is shown below:

```
starting runs:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
poll 0  memory 60.45 MB
  Entering ./runs/diamond/scf 0
    writing input files  0 scf
  Entering ./runs/diamond/scf 0
    sending required files  0 scf
    submitting job  0 scf
  Entering ./runs/diamond/scf 0
    Would have executed:
      export OMP_NUM_THREADS=1
      mpirun -np 16 pw.x -input scf.in

poll 1  memory 60.72 MB
  Entering ./runs/diamond/scf 0
    copying results  0 scf
  Entering ./runs/diamond/scf 0
    analyzing  0 scf

poll 2  memory 60.73 MB
  Entering ./runs/diamond/scf 1
    writing input files  1 conv
  Entering ./runs/diamond/scf 1
    sending required files  1 conv
    submitting job  1 conv
  Entering ./runs/diamond/scf 1
    Would have executed:
      export OMP_NUM_THREADS=1
      mpirun -np 1 pw2qmcpack.x<conv.in

poll 3  memory 60.73 MB
  Entering ./runs/diamond/scf 1
    copying results  1 conv
  Entering ./runs/diamond/scf 1
    analyzing  1 conv

poll 4  memory 60.73 MB
  Entering ./runs/diamond/vmc 2
    writing input files  2 vmc
  Entering ./runs/diamond/vmc 2
    sending required files  2 vmc
    submitting job  2 vmc
  Entering ./runs/diamond/vmc 2
    Would have executed:
      export OMP_NUM_THREADS=4
```

```
        mpirun -np 4 qmcapp vmc.in.xml

  poll 5  memory 60.78 MB
    Entering ./runs/diamond/vmc 2
      copying results  2 vmc
    Entering ./runs/diamond/vmc 2
      analyzing  2 vmc

Project finished
```

The output describes the progress of each simulation. The run submission commands are also clearly shown as well as the amount of memory used by Nexus. There should now be a "`runs`" directory containing the generated input files with the following structure:

```
runs/
\-- diamond
    |-- scf
    |-- |-- C.BFD.upf
    |-- |-- conv.in
    |-- |-- pwscf_output
    |-- |-- scf.in
    |-- |-- sim_conv
    |-- |-- |-- input.p
    |-- |-- \-- sim.p
    |-- \-- sim_scf
    |--     |-- input.p
    |--     \-- sim.p
    \-- vmc
        |-- C.BFD.xml
        |-- sim_vmc
        |-- |-- input.p
        |-- \-- sim.p
        \-- vmc.in.xml
```

The "`sim.p`" files record the state of each simulation. Inspect the input files generated by Nexus (`scf.in`, `conv.in`, and `vmc.in.xml`). Compare the files with the input provided to Nexus in `diamond.py`.

Finally, run the example in execute mode. Remove the "`runs`" and "`results`" directories, set "`status_only`=0" and "`generate_only`=0", and rerun the script. The output shown should be similar to what was seen for generate mode, only now there may be multiple workflow polls (see section 3) while a particular simulation is running. The "`sleep`" keyword controls how often the polls occur (every 3 seconds in this example). Note that the

Nexus host process sleeps in between polls so that a minimum of computational resources are occupied. Once "`Project finished`" is displayed, all the simulation runs should be complete. Confirm the success of the runs by checking the output files. The text "`JOB DONE.`" should appear near the end of the PWSCF output file `scf.out`. QMCPACK has completed successfully if "`Total Execution time`" appears near the end of the output in `vmc.out`.

7. Summary

In this paper a new automation system for simulation workflows has been described. The Nexus package allows users to compose both simple and complex simulation workflows tailored to their needs and execute them on diverse computing resources within a common accessible framework. Use of the system offers productivity gains, improved reproducibility, and provenance of full scientific workflow procedures in addition to the generated data. Quantum simulation codes that are currently supported include QMCPACK, Quantum Espresso, VASP, and GAMESS. Further additions are anticipated in the near future.

8. Acknowledgements

The author would like to thank Jeongnim Kim, Kateryna Foyevtsova, Juan Santana-Palacio, Chandrima Mitra, Ryan McAvoy, Hemant Dixit, and the participants of the ALCF-hosted 2014 QMC Training Program for working with Nexus during its development. The author is indebted to Paul Kent for a thorough reading of the manuscript. This work was primarily supported through Predictive Theory and Modeling for Materials and Chemical Science program by the Office of Basic Energy Sciences (BES), Department of Energy (DOE). Initial work was supported by the National Science Foundation (NSF) under contract OCI-0904572 at the Department of Physics of the University of Illinois at Urbana-Champaign.

Appendix A. Structure operations

Listed below are a sampling of the operations made available to the user through the Structure class.

General operations:

- convert internal representation between different distance unit systems

Cell operations:

- rescale or skew the simulation cell to introduce strain
- center molecules and solids within the simulation cell
- create a bounding box surrounding a molecule
- calculate simulation cell volume
- calculate Wigner and inscribing radii of simulation cell
- compute electropotential Madelung constant of a cell (useful for Makov-Payne corrections)

Structure operations:

- generate crystal structures following the major space groups following a lattice+basis approach
- expand a structure in repeated fashion by tiling by integer multiples along cell axes or by full reshaping of the cell (e.g. from fcc to simple cubic by matrix tiling)
- cleave bulk crystals to form surfaces
- embed a smaller relaxed structure within a larger unrelaxed one
- obtain atomic positions in units of cell axes
- obtain atomic species and atoms grouped by species
- calculate distances between atoms in related structures
- calculate distances or difference vectors in the minimum image convention
- calculate scalar or vector distance tables and neighbor tables
- find atoms that are outermost in a molecule or structure (convex hull)

- determine the sequence of nearest neighbor shells atoms surrounding a selected group of atoms
- locate or select groups of atoms by index, atomic species, mask array or within a subcell, spheres surrounding a set of points or selected group of atoms or the negation/complement of any of these
- freeze selected atoms from moving in particular directions
- remove selected atoms to create vacancies or voids
- replace selected atoms by others to create more complicated defects
- read atomic species and positions from XYZ and POSCAR files
- write atomic species and positions to XYZ and xsf files

K-point operations:

- add a Monkhorst-Pack grid of k-points
- obtain a mapping of supercell to primitive cell kpoints
- obtain k-points in units of reciprocal cell axes

[1] J. Yu, R. Buyya, A taxonomy of scientific workflow systems for grid computing, SIGMOD Rec. 34 (3) (2005) 44–49. doi:10.1145/1084805.1084814.
URL http://doi.acm.org/10.1145/1084805.1084814

[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems (0) (2014) –. doi:http://dx.doi.org/10.1016/j.future.2014.10.008.
URL http://www.sciencedirect.com/science/article/pii/S0167739X14002015

[3] C. Ortiz, O. Eriksson, M. Klintenberg, Data mining and accelerated electronic structure theory as a tool in the search for new functional materials, Computational Materials Science 44 (4) (2009) 1042 – 1049.

doi:http://dx.doi.org/10.1016/j.commatsci.2008.07.016.
URL http://www.sciencedirect.com/science/article/pii/S0927025608003534

[4] J. Hachmann, R. Olivares-Amaya, S. Atahan-Evrenk, C. Amador-Bedolla, R. S. Sánchez-Carrera, A. Gold-Parker, L. Vogt, A. M. Brockway, A. Aspuru-Guzik, The harvard clean energy project: Large-scale computational screening and design of organic photovoltaics on the world community grid, The Journal of Physical Chemistry Letters 2 (17) (2011) 2241–2251. arXiv:http://dx.doi.org/10.1021/jz200866s, doi:10.1021/jz200866s.
URL http://dx.doi.org/10.1021/jz200866s

[5] 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, Computational Materials Science 58 (0) (2012) 218 – 226. doi:http://dx.doi.org/10.1016/j.commatsci.2012.02.005.
URL http://www.sciencedirect.com/science/article/pii/S0927025612000717

[6] J. S. Hummelshøj, F. Abild-Pedersen, F. Studt, T. Bligaard, J. K. Nørskov, Catapp: A web application for surface chemistry and heterogeneous catalysis, Angewandte Chemie International Edition 51 (1) (2012) 272–274. doi:10.1002/anie.201107947.
URL http://dx.doi.org/10.1002/anie.201107947

[7] A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, K. A. Persson, Commentary: The materials project: A materials genome approach to accelerating materials innovation, APL Materials 1 (1) (2013) –. doi:http://dx.doi.org/10.1063/1.4812323.
URL http://scitation.aip.org/content/aip/journal/aplmater/1/1/10.1063/1.4812323

[8] J. Krüger, R. Grunzke, S. Gesing, S. Breuers, A. Brinkmann, L. de la Garza, O. Kohlbacher, M. Kruse, W. E. Nagel, L. Packschies,

R. Müller-Pfefferkorn, P. Schäfer, C. Schärfe, T. Steinke, T. Schlemmer, K. D. Warzecha, A. Zink, S. Herres-Pawlis, The mosgrid science gateway a complete solution for molecular simulations, Journal of Chemical Theory and Computation 10 (6) (2014) 2232–2245. arXiv:http://dx.doi.org/10.1021/ct500159h, doi:10.1021/ct500159h.
URL http://dx.doi.org/10.1021/ct500159h

[9] W. M. C. Foulkes, L. Mitas, R. J. Needs, G. Rajagopal, Quantum monte carlo simulations of solids, Rev. Mod. Phys. 73 (2001) 33–83. doi:10.1103/RevModPhys.73.33.
URL http://link.aps.org/doi/10.1103/RevModPhys.73.33

[10] G. v. Rossum, et al., Python programming language, URL http://www.python. org.

[11] J. Kim, K. P. Esler, J. McMinis, M. A. Morales, B. K. Clark, L. Shulenburger, D. M. Ceperley, Hybrid algorithms in quantum monte carlo, J. Phys. Conf. Ser. 402 (1) (2012) 012008.
URL http://stacks.iop.org/1742-6596/402/i=1/a=012008

[12] C. J. Umrigar, J. Toulouse, C. Filippi, S. Sorella, R. G. Hennig, Alleviation of the fermion-sign problem by optimization of many-body wave functions, Phys. Rev. Lett. 98 (2007) 110201.

[13] W. McMillan, Ground state of liquid he4, Phys. Rev. 138 (2A) (1965) A442–A451.

[14] R. Grimm, R. Storer, Monte-carlo solution of schrödinger's equation, J. Comput. Phys. 7 (1) (1971) 134–156.

[15] J. B. Anderson, A random-walk simulation of the schr[o-umlaut]dinger equation: H[sup +][sub 3], J. Chem. Phys. 63 (4) (1975) 1499–1503. doi:10.1063/1.431514.
URL http://link.aip.org/link/?JCP/63/1499/1

[16] V. Fock, Nherungsmethode zur lsung des quantenmechanischen mehrkrperproblems, Zeitschrift fr Physik 61 (1-2) (1930) 126–148. doi:10.1007/BF01340294.
URL http://dx.doi.org/10.1007/BF01340294

[17] C. C. J. Roothaan, New developments in molecular orbital theory, Rev. Mod. Phys. 23 (1951) 69–89. doi:10.1103/RevModPhys.23.69.
URL http://link.aps.org/doi/10.1103/RevModPhys.23.69

[18] P. G. et. al., QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials, Journal of Physics: Condensed Matter 21 (39) (2009) 395502 (19pp).
URL http://www.quantum-espresso.org

[19] W. Kohn, L. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (4A) (1965) A1133–A1138.

[20] P. Hohenberg, W. Kohn, Inhomogeneous electron gas, Phys. Rev. 136 (3B) (1964) B864.

[21] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, J. A. Montgomery, General atomic and molecular electronic structure system, Journal of Computational Chemistry 14 (11) (1993) 1347–1363. doi:10.1002/jcc.540141112.
URL http://dx.doi.org/10.1002/jcc.540141112

[22] M. S. Gordon, M. W. Schmidt, Advances in electronic structure theory: GAMESS a decade later, Elsevier, Amsterdam, 2005, pp. 1167–1189.

[23] G. Kresse, J. Hafner, Ab initio molecular dynamics for liquid metals, Phys. Rev. B 47 (1993) 558.

[24] G. Kresse, J. Hafner, Ab initio molecular-dynamics simulation of the liquid-metal-amorphous-semiconductor transition in germanium, Phys. Rev. B 49 (1994) 14251.

[25] G. Kresse, J. Furthmüller, Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set, Comput. Mat. Sci. 6 (1996) 15.

[26] 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.

[27] E. Walter, OPIUM, http://opium.sourceforge.net.

[28] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible markup language (xml), World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210 (1998) 16.

[29] M. Folk, A. Cheng, K. Yates, Hdf5: A file format and i/o library for high performance computing applications, in: Proceedings of Supercomputing, Vol. 99, 1999, pp. 5–33.

[30] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, D. Tweten, Portable batch system: External reference specification, Tech. rep., Technical report, MRJ Technology Solutions (1999).

[31] J. Krogel, J. Kim, D. Ceperley, Prospects for efficient qmc defect calculations: the energy density applied to ge self-interstitials, in: APS Meeting Abstracts, Vol. 1, 2013, p. 24005.

[32] K. Foyevtsova, J. T. Krogel, J. Kim, P. Kent, E. Dagotto, F. A. Reboredo, *Ab initio* quantum monte carlo calculations of spin superexchange in cuprates: The benchmarking case of ca_2cuo_3, Phys. Rev. X 4 (2014) 031003. doi:10.1103/PhysRevX.4.031003.
URL `http://link.aps.org/doi/10.1103/PhysRevX.4.031003`

[33] J. T. Krogel, J. Kim, F. A. Reboredo, Energy density matrix formalism for interacting quantum systems: Quantum monte carlo study, Phys. Rev. B 90 (2014) 035125. doi:10.1103/PhysRevB.90.035125.
URL `http://link.aps.org/doi/10.1103/PhysRevB.90.035125`

[34] J. A. Santana, J. T. Krogel, J. Kim, P. R. Kent, F. A. Reboredo, it Ab-initio many-body calculations of the oxygen vacancy in zno, arXiv preprint arXiv:1406.3169.

[35] C. Mitra, J. Krogel, J. A. Santana Palacio, F. A. Reboredo, Quantum monte carlo calculations of structural and electronic properties in the correlated oxide nio, Bulletin of the American Physical Society 60.

[36] J. Krogel, J. Santana, P. Kent, F. Reboredo, Pseudopotentials for quantum monte carlo calculations of transition metal oxides, Bulletin of the American Physical Society 60.

[37] H. Dixit, J. H. Lee, J. T. Krogel, S. Okamoto, V. R. Cooper, Stabilization of weak ferromagnetism by strong magnetic response to epitaxial strain in multiferroic bifeo3, manuscript submitted for publication. (2015).

[38] M. Burkatzki, C. Filippi, M. Dolg, Energy-consistent pseudopotentials for quantum monte carlo calculations, The Journal of Chemical Physics 126 (23) (2007) 234105. doi:10.1063/1.2741534.
URL http://link.aip.org/link/?JCP/126/234105/1