Developing Interoperable, Accessible Software via the Atomic, Molecular, and Optical Sciences Gateway: A Case Study of the B-spline atomic R-Matrix code Graphical User Interface

Tom Wolcott[1,2], Klaus Bartschat[3], Barry I. Schneider[2], Sudhakar Pamidighantam[4], and Kathryn R. Hamilton[5]
[1]*University of Maryland, College Park, MD 20742, USA*
[2]*Applied and Computational Mathematics Division, NIST, Gaithersburg, MD 20899, USA*
[3]*Department of Physics and Astronomy, Drake University, Des Moines, IA 50311, USA*
[4]*Center for Artificial Intelligence in Science and Engineering, Institute for Data Engineering and Science, Georgia Institute of Technology, North Avenue Atlanta, GA 30332, USA*
[5]*Department of Physics, University of Colorado Denver, Denver, CO 80204, USA*
(Dated: May 2024)

The Atomic, Molecular, and Optical Science Gateway [1] is a comprehensive cyberinfrastructure for research and educational activities in computational AMO science. The *B*-Spline atomic *R*-Matrix (BSR) suite of programs is one of several computer programs currently available on the gateway. It is an excellent example of how the gateway increases the scientific productivity of AMOS users. While the suite is available to be used in batch mode, its complexity does not make it well-suited to the approach taken in the gateway's default setup. The complexity originates from the need to execute many different computations and to construct generally complex workflows, requiring numerous input files that must be used in a specific sequence. The BSR graphical user interface (GUI) described in this paper was developed to considerably simplify employing the BSR codes on the gateway, making BSR available to a large group of researchers and students interested in AMO science.

I. INTRODUCTION

The field of Atomic, Molecular, and Optical Science (AMOS) has a rich history of using computational methods to tackle pressing research questions. As well as providing support and guidance to experimental efforts, in some sub domains, such as electronic structure and electron scattering, computational approaches are the most prevalent, and most successful, ways of performing AMOS research. Despite this widespread adoption of computation, the field of AMOS suffers from a lack of code sharing and standardization efforts. Most AMOS groups develop their own software in-house, resulting in small user bases, duplication of efforts across groups, difficulty in comparing the outputs from different codes even when tackling the same physical problem, and software that is burdensome for novice external scientists to use.

A recent effort seeking to address this situation is the Atomic, Molecular, and Optical Science (AMOS) Gateway [1, 2]. Following from a 2019 workshop at the Institute for Theoretical Atomic and Molecular Physics at the Harvard Smithsonian Center for Astrophysics, a number of active researchers in computational atomic and molecular physics grouped together to create the AMOSGateway, which is now entering a rapid stage of development thanks to support from the National Science Foundation through a Cyberinfrastructure for Sustained Scientific Innovation award. The overarching goal of the AMOS Gateway is to create a comprehensive cyberinfrastructure for the AMOS community, where practitioners can access a synergistic, full-scope platform for computational AMOS. Software suites are contributed by AMOS researchers, then compiled on a number of NSF-supported computational platforms, and an interface is created linking this deployment to the AMOSGateway website. This means that using the AMOS Gateway does not require the user to download and build these codes on their own platform, which is often difficult, especially for non-experts. Instead, the user starts with appropriate data input files for the gateway software interfaces, or takes and/or modifies example inputs provided and ready for execution, submits the calculation via the gateway interface, and the results are returned on the gateway. Data may then be analyzed *in situ* or downloaded to another platform for more detailed inspection. Recent descriptions of the AMOS Gateway can be found in [3, 4].

In order to achieve the goal of making the AMOS Gateway useful as both a research and an educational tool for practitioners and researchers interested in computational AMOS, the AMOS Gateway team is investing significant effort in the creation and development of user interfaces for each hosted software suite. Our overarching design principles are that the interfaces must be 1) intuitive for novice users, 2) maintain the functionality desired by advanced users, 3) provide efficient ways to perform complex, multi-step calculations, and 4) allow future interoperability between software suites. Of the ten currently-hosted software suites, development of these new interfaces has begun for two codes: time-dependent Recursive indeXing (tRecX) and the B-Spline atomic R-Matrix code (BSR).

In this paper, we will use the updated BSR interface as a case study in interoperable and accessible software development via the AMOS Gateway. We first give a brief overview of the BSR code and its operation in traditional batch mode, before presenting the new AMOS Gateway BSR user interface and commenting on our design philosophy and process. We will then describe the implementation of workflows into the BSR user interface and detail how these could be used in the future to perform multi-step calculations using several different BSR calculations, or several different software suites. Finally, we will comment on future plans for both the BSR interface and the larger AMOS Gateway ecosystem.

This case study is expected to be useful to the Chemical Physics community for two reasons. Firstly, computational chemical physicists are end-users of data generated by codes on the AMOS Gateway. Hence, they may be interested in how these data are produced and how to possibly create such data themselves. Secondly, members of the computational chemical physics community may wish to develop science gateways themselves, and so can draw inspiration from this example.

II. THE B-SPLINE ATOMIC R-MATRIX CODE

One important illustration of the kind of physical problems the AMOS gateway can treat involves the quantum-mechanical description of collisions of atoms and molecules with electrons and photons. Treatment of these processes provides insights into elastic and inelastic collisions, ionization, atomic and molecular structure and stability, which are essential for understanding plasma processes, lasers, and generating data to interpret astrophysical observations. Among the many currently available approaches to treat such problems, the R-Matrix method [5] is significant, as it treats the problem as a *general* $(N+1)$-electron system with an N-electron target and a colliding electron. Non-relativistic, semi-relativistic (Breit-Pauli), and even full-relativistic (at the Dirac-Breit level) formulations and associated computer codes exist. For practical applications, the most-frequently employed version is the Belfast suite of codes, as published many years ago [6] with updates available through various websites.

The B-Spline R-Matrix (BSR) method discussed in this paper was introduced by Zatsarinny and Froese Fischer [7]. It was first applied as a proof-of-principle demonstration to photoionization of atomic lithium. The method was then further developed by Zatsarinny who published his general computer code in 2006 [8]. The principal difference with the Belfast codes mentioned above is the use of different sets of non-orthogonal orbitals to represent both the bound and continuum one-electron functions and to employ a set of B-splines as the R-Matrix basis functions. The term-dependent, and hence non-orthogonal, bound orbital sets allow a much higher accuracy in the description of the target states with small configuration-interaction expansions than orthogonal bases, and the finite-element B-splines provide computational flexibility due to their excellent numerical approximation properties that are far superior compared to finite-difference approaches.

For the past two decades, Zatsarinny's code has been applied to numerous atomic structure and collision problems, including processes induced by weak-field continuous and strong-field, short-pulse electromagnetic radiation, i.e., bound-bound and bound-continuum transitions. The results are generally considered as benchmarks to be checked against for predictions from any alternative method. For a description of the basic theory, as well as an extensive list of examples and early applications, we refer to the review by Zatsarinny and Bartschat [9]. The software suite is also publicly available on Github [10].

The BSR suite of codes is summarized in Fig. 1. It can be split into three categories: modules, utilities, and workflows. Modules are units of a larger program that carry out a specific operation. For example, `BSR_CONF` computes the close-coupling expansions for later steps, while `BSR_HD` performs the final matrix diagonalization. Utilities are designed to carry out specific stand-alone tasks, such as converting file types or to obtain selected observables from the general output, such as cross sections from the transition-matrix elements. Workflows then connect the inputs and outputs of these individual programs and utilities in order to carry out entire computations. For example, the `STGF` workflow ingests information from the scattering workflow and produces electron scattering cross sections.

In addition to the variety of programs that can be run, each module has its own set of input files, usually generated by previous modules in the calculation sequence. There are also common input files, which are required at the start of every BSR calculation. These are the *target* file and associated *name.c* and *name.bsw* files, which contain orbital information that may be updated further by some of the programs executed during the workflow. The *bsr_par* file contains specific numerical and calculation parameters, and the *knot.dat* file describes the B-Spline grid. A full description of each module, input and output files, and main

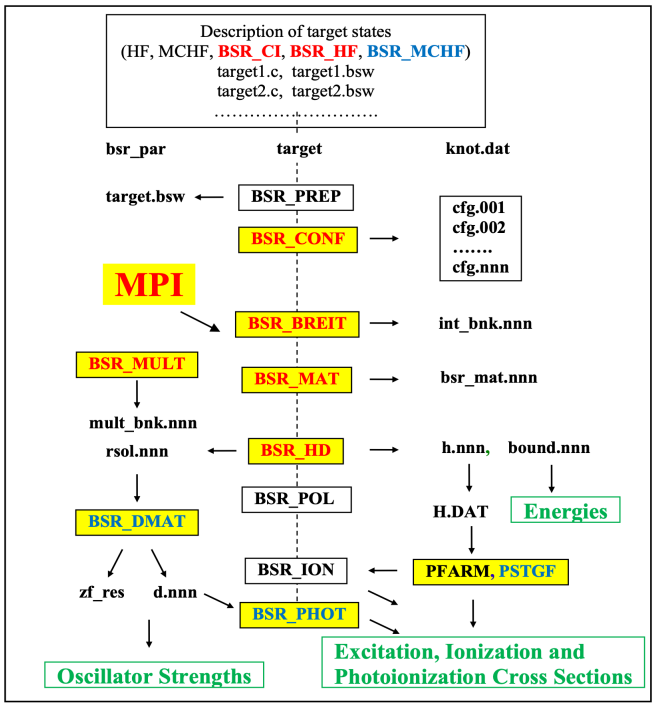

FIG. 1. Structure of the BSR software suite, showing BSR modules (BSR_X, contained in boxes), input files, output files (x.nnn), and main observables (green boxes). Observables are obtained from the various output files by additional utility codes, which are not included in this schematic. Multiple modules and utilities must therefore be run in the correct order to generate the desired observables. These combinations are referred to as workflows.

utilities can be found in Ref. [8].

The multiple libraries, modules, and utilities make compilation of the BSR suite relatively complex, and preparing the necessary inputs, executing the modules and utilities in the correct order, and handling the multiple output files further complicates performing BSR calculations. While one could prepare PERL or BASH scripts to create workflows that would simplify the execution process, it is difficult to make these sufficiently general, and this still does not alleviate the burden of compilation. Experienced users may also wish to use advanced functionalities of the BSR suite. This typically involves adjustments to parameters and input files between the execution of modules, which is difficult to allow in a traditional script. Additionally, high-performance computing resources are typically required to perform any reasonably-sized scattering calculation within an acceptable timeframe. The AMOS Gateway easily handles both the first and last problem by making pre-compiled software available to users on a variety of HPC systems along with computing time on these resources. The main challenge behind the AMOS Gateway development is addressing the middle two concerns: how does one create a user interface that maintains sufficient functionality for experienced users, but is still accessible to novices who wish to perform their first calculations?

III. BSR USER INTERFACE DESIGN

The design philosophy behind the BSR interface largely follows that of the tRecX interface [11] whose goal was to not only be useful for experienced users to setup multiple runs with dynamically extendable user input interfaces, but also to be made accessible for novice users with embedded help and default values for many simulation parameters. This played into many parts of the BSR interface design process, including deciding between multiple revisions of the input selector and ensuring similarities between the two interfaces.

In addition, the design for tRecX centered around the concept of a "Run", which represents a computation with its inputs, with multiple runs for potentially different parameters that can be compared as a set. This led to designs for the user to create the run, set it to be executed, monitor its progress, and view its outputs. Alongside the runs, "Views" allow users to group together a set of runs. These concepts were brought directly into the BSR interface, not only to save work, but also to provide consistency in user interfaces across applications. Again, because of this, a conscious effort was made to avoid changing the meanings of terms leveraged from the tRecX user interfaces.

The design for the BSR "Run" page is shown in Fig. 2. It was important to be able to carry out some operations on many runs at a time: saving them into a view, comparing them, and deleting them. Along with this, each run has associated action `buttons`, which allow the user to quickly access copying and deleting the run.

However, the "Create new run" page required a different approach due to differences in BSR's many input files and

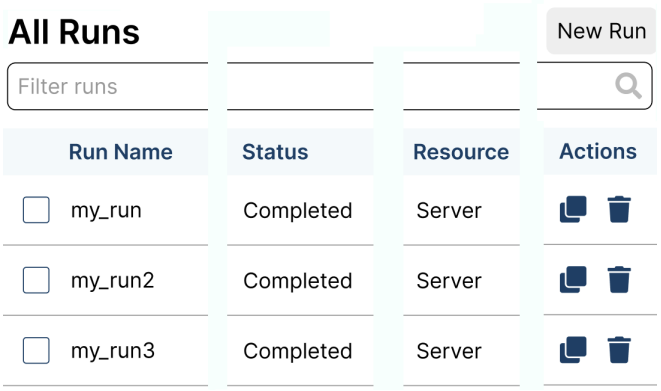

FIG. 2. Design of the BSR runs page. The user can select runs to save them to a view, quickly view run status, and copy and delete runs.

types of programs to be executed as workflows. The design of the page, therefore, was centered around the run type/input selector, whose design required that the user should be able to select from any of the computations organized into the aforementioned categories of modules, utilities, and workflows. In addition, since each of these computations had their own inputs, the user needed to be able to see what inputs are required and which of those must be uploaded/satisfied. Multiple variations for this element were designed with the goal of compactness and the ability to show the current state of all inputs. The design employed `buttons` for both selecting the computation and the appropriate files for viewing/editing. During the implementation, the design was slightly altered as shown in Fig. 3 in order to clearly distinguish between the inputs and the run type, and to better communicate the state of the input with different icons.

Another part of the design was the table view as seen in Fig. 4. In order for the interface to be friendly to new users, it is important that they have a graphical user interface with input validation and labels as opposed to just a text view. This is accomplished by a tab selector above the file editor, which switches between a text mode, table mode, and graphing mode. When a file supports any of the options, the button will be enabled and the user can opt to switch between them. If a file supports a table view, that option is automatically selected when the file is opened. The table view provides the information under individual tabs to organize the input data into sections and reduce the complexity.

The program is deployed via a Github webhook that runs on the creation of a new github tag. A webhook is an HTTP-based call-back program that provides event-driven communication between two programs or event triggered execution of a second program as part of an automated workflow. A webhook is registered at the git repository with a specific **secret** that sends the trigger message. A webhook daemon on the gateway server listens for this message and, once the message is received, triggers the update code. The BSR-Update triggering webhook configuration is presented in listing 1. The execution of the `bsr-update` script is

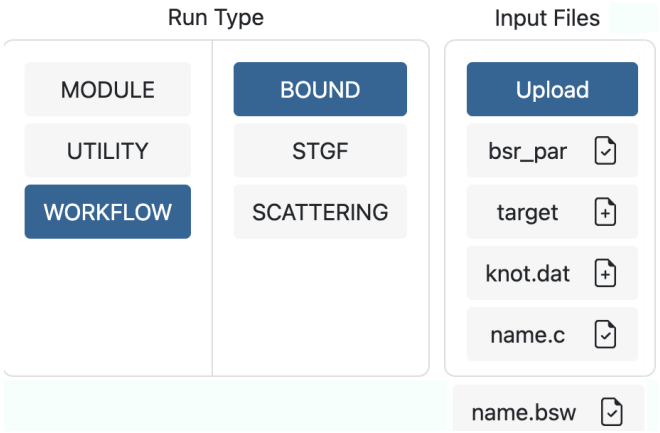

FIG. 3. The current design of the run type/input selector. All run type options are displayed in the left column, with the currently selected run type highlighted in blue. Additional options for the selected run type are displayed in the middle column. Once a run type and additional options have been selected, required input file names are displayed in the input file selector. Uploaded / selected input files have an icon of a file with a check mark displayed to their right, required files that have yet to be uploaded have an icon of a file with a plus sign displayed to their right.

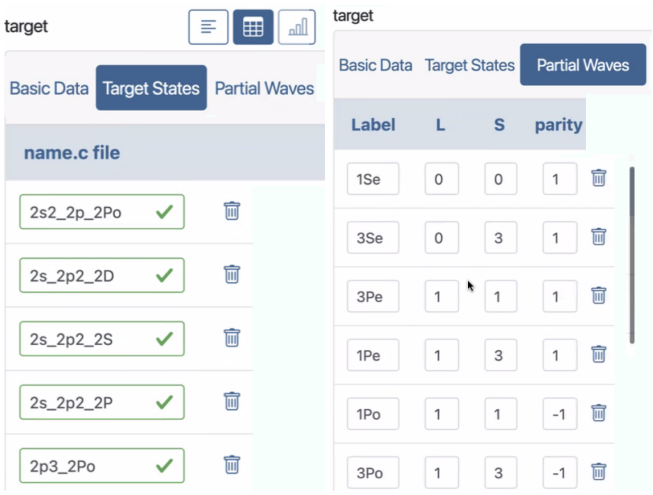

FIG. 4. Table view for editing/viewing a *target* file, one of the required BSR input files. The *target* file contains three key sections: "Basic Data" related to the number of electrons and coupling scheme; "Target States" with the names of the *.c* and *.bsw* files that provide the atomic structure descriptions of the N-electron system (shown on the left), and "Partial Waves" that describe the $(N+1)$-electron symmetries.

then triggered.

The Auto Update script is depicted in listing 2. These scripts are executed in a controlled virtual environment. The update carries out a git pull to refresh the BSR interface code from the git repository, installs any required dependencies, and then builds the BSR Django application using the Yarn package and build manager. Once the code is ready, instructions to propagate changes in models (adding a field, deleting a model, etc.) into a Django local database schema are executed using the Django migrate function. The Django `collectstatic` function updates the node modules and applications. Finally, a touch command will refresh the Django application to provide the new interface pages in the gateway. Currently, we use this procedure to deploy and test the newly developed user interfaces in a developmental gateway and once finalzied, a separate process updates the production gateway.

Listing 1. Webhook configuration for BSR Update deployment. See text for details.

```
{
  "id": "bsr-update",
  "execute-command": "/g/pga/webhooks/
      commands/bsr-update",
  "pass-arguments-to-command": [ {
      "source": "payload",
      "name": "repository.name" } ],
  "trigger-rule": {
    "and": [ {
        "match": {
          "type": "payload-hash-sha1",
          "secret":
              "******************",
          "parameter":  {
            "source": "header",
            "name": "X-Hub-Signature" }
              } } ]  }
}
```

Listing 2. BSR Auto Update. See text for details.

```
# Instead of pip install, manually update
    the django app
cd /var/www/portals/django-amp/
    bsr_django_app/
git pull
pip install -e .
cd bsr_django_app
yarn
yarn build

cd /var/www/portals/django-amp/airavata-
    django-portal/
python manage.py migrate
python manage.py collectstatic -i
    node_modules --noinput
touch django_airavata/wsgi.py
```

IV. IMPLEMENTATION

For the BSR user interface, the Django web Framework was used to interface with the Airavata Django Portal Framework (ADPF) [12] to provide REST APIs for the Vue [13] front end. These technologies were chosen to

match the software stack used in the rest of the Airavata ecosystem. In order to match the visual style seen in the rest of the website, Bootstrap Vue [14] components were employed where possible and other components were developed as needed. The structure of the entire program, from the use of the Django REST framework to Vuex [15] state management and Bootstrap Vue, were adapted from the tRecX interface development [11].

In the back end, Django's object relational model (ORM) was used to store data as Models with the same general structure as tRecX. A brief description of these models is contained in Table I. In order to connect these models to the front end, the Django REST framework was used to set up APIs using its `ModelSerializer` and `ModelViewSet`. Since the `ViewSets` allow posting/updating/getting models directly through the API, it was used to connect the views, runs, and plot parameters as described in the Table II. By default any `ModelViewSets` implement the following actions through the API: `list`, `create`, `retrieve`, `update`, `partial_update`, `destroy`. These can be overwritten if needed.

Model	Description
Run	Used to represent a prepared computation with its inputs.
Input	Connected to a run, it represents an input that can be a parameter, files, or the run type. Closely interfaces with the Airavata application interface.
File	Connected to an input, represents a file uploaded to a run.
Remote Execution	Connected to a run, represents an execution of the run.
View	Connected to multiple runs, represents some logical grouping of runs.
Plot Parameters	Holds plot parameters for the plotting of a file, so they can be reused.

TABLE I. The Django models used for the updated BSR interface.

On the front end the Django REST API is called using Axios [16], an HTTP client. The calls to the API are grouped within objects that are exported from a JavaScript module as services. These services also ensure that the objects returned are properly encoded in the expected format both to and from the API. This allows the services to switch field names between snake to camel case, remove irrelevant fields, convert data, etc. While these services can be called anywhere in the program, it is often more useful to store the fetched values in a single Javascript module. Vuex, a state management library for Vue, accomplishes this through Vuex stores. Each store consists of a state, commit methods, getters, and actions; commit methods edit the state, getters read the state, and actions perform tasks like fetching views or submitting runs. This allows the state of the interface to be centralized within the Vuex store, freeing each page from having to carry the entire state.

The pages for the interface are routed on the front end with vue-router [17], which provides the `router-view` component that controls client-side routing by updating its content based on the pages's URL. This is used in the main App.vue component to display several pages that make up the interface, each page being composed of several Vue components. This allows pages to be broken down into their base components, which can then be reused. Figure 5 shows the breakdown of part of the "Create new run" page, to illustrate this organization. Inside the page, two `Badge` components are used to display the Experiment and Job statuses of a run. The color of the badge changes to help communicate the status of the run. For example, when the run is completed the badge will be green, and if it fails it will be red. There is also the `PathSelector`, which selects the file to be viewed. This is then separated, for the sake of clarity, into two `buttonsGroup` components. There is also the `FileOptions` component that contains various views for a file and allows the user to switch between them. In the figure, the table view is selected. This brings up the `TableView` component in which each input uses the `DataInput` component that holds the label as a header, validation checks, and value for the given input.

A. Input Handling

Due to BSR's many input files, allowing users to upload all files at once is a critical feature. However, categorizing, validating and handling the input files is quite complex. Some BSR modules require a single file, while others require multiple. Typically, whenever a computation requires many of a particular file type, there will be an index in the filename with the rest of the name matching (e.g., *bound.nnn* files, where *nnn* is a three-digit number). This comes with a few exceptions, however. For example, the *tr_nnn_nnn* files have two distinct indices while the *.c* and *.bsw* files have their own unique names. When all files are uploaded the code in Listing 3 is run. This function maps the name of a file to the name of the corresponding input. To do this, each filename is checked against valid input names. If the name matches, it returns the name. Otherwise it will try to replace any indices in the file name with *nnn* to check if the name with *nnn* replaced matches. If not, it is checked whether the file is a *.c* or a *.bsw* file.

Even though this process matches all valid file inputs, uploading a file with a mismatched name to a specific input is still possible. This matching also maps the filename to its required input name. For a single file, the input name is the same, but for an input with multiple files such as *name.c* with changing *name* or *bound.nnn* with changing *nnn* the matching becomes slightly more complicated. Listing 3 shows pseudocode to resolve this.

Since the structure and layout of BSR's files vary, the code that handles the specifics of each file is kept within the *fileData.js* module. In this module, the input name is used as a key to access data for the file and the specifics of how to process it. The exported objects are listed below:

- **descriptions**: Hold text descriptions of the run types and inputs, which are displayed to the user to help them work

ModelViewSet	API call	Description
Run	defaults	
	create	Creates a directory for the files of the run alongside input and file models, this then returns the resulting run model.
	update	Updates the run model and any files or inputs that have been changed.
	destroy	Deletes the model and all input and file models associated with it. In addition, a boolean delete_associated is passed in which, if set to true, also deletes the run's directory in BSR_Runs.
	get_output_files	Since BSR has many different outputs from its various programs, this call returns all of the runs outputs, obtained from the ARCHIVE directory in its experiment directory.
	submit	Collects all the inputs into a dictionary in order to set the experimentInputs on a newly created ExperimentModel.
View	defaults	
	create	Adds itself to the runs' views with the specified runIds
	update	If override=true is passed into the call, the view will be removed from runs that it is no longer attached to, regardless of whether the view is added to runs in runIds.
PlotParameters	defaults	
	plot	With the plot files and the plotting parameters, it runs the Python script *plot.py* to generate the graph, which is returned as a base64-encoded image
	api_settings	Returns the application id for BSR.

TABLE II. The API setup for the updated BSR interface. See text for details.

Listing 3. Pseudocode for matching filenames

```
FUNCTION getInputName(filename, inputNames
    )
    IF filename is in inputNames THEN
        RETURN filename
    END IF

    LET nnnReplacedFilename =
        filename.replaceAll(/\\d{3}/g, "
            nnn")

    IF nnnReplacedFilename is in
        inputNames THEN
        RETURN nnnReplacedFilename
    END IF

    IF filename ends with ".c" THEN
        RETURN "name.c"
    END IF

    IF filename ends with ".bsw" THEN
        RETURN "name.bsw"
    END IF

    RETURN null
END FUNCTION
```

with the files.

- **plotObjects**: Contain the x-axes and y-axes available for plotting.
- **tableObjects**: Contain an object that describes how to create a table for its input. The table object is structured into pages that split the file into parts. Each page is then either displayed as a table of values or as a list of key value pairs as shown in Figs. 4 and 6. These objects use regular expressions on the text to get and set data in place, which avoids issues with keeping the text and table in sync.

The selection of compute resources uses the same Vue RunResource component from tRecX, which makes use of the ADPF components provided through Airavata; see Fig. 7. These allow the user to select how the computation will be allocated alongside other options. Once the run is saved, the settings are stored inside the Run model in the Django back end.

B. Status Updates

Through the course of using the interface, it is important that the user is kept aware of the state that the interface is in. For this reason, it was important to show loading screens and error messages. The code snippet below, Listing 4, was a common pattern used in the code to combine both error handling and the use of loading screens, since API calls often required being wrapped in both.

To display errors to the user, an event bus was employed in order to allow any part of the program to surface an error. Wherever an important error might occur, the code was wrapped in a try-catch block with a caught error being

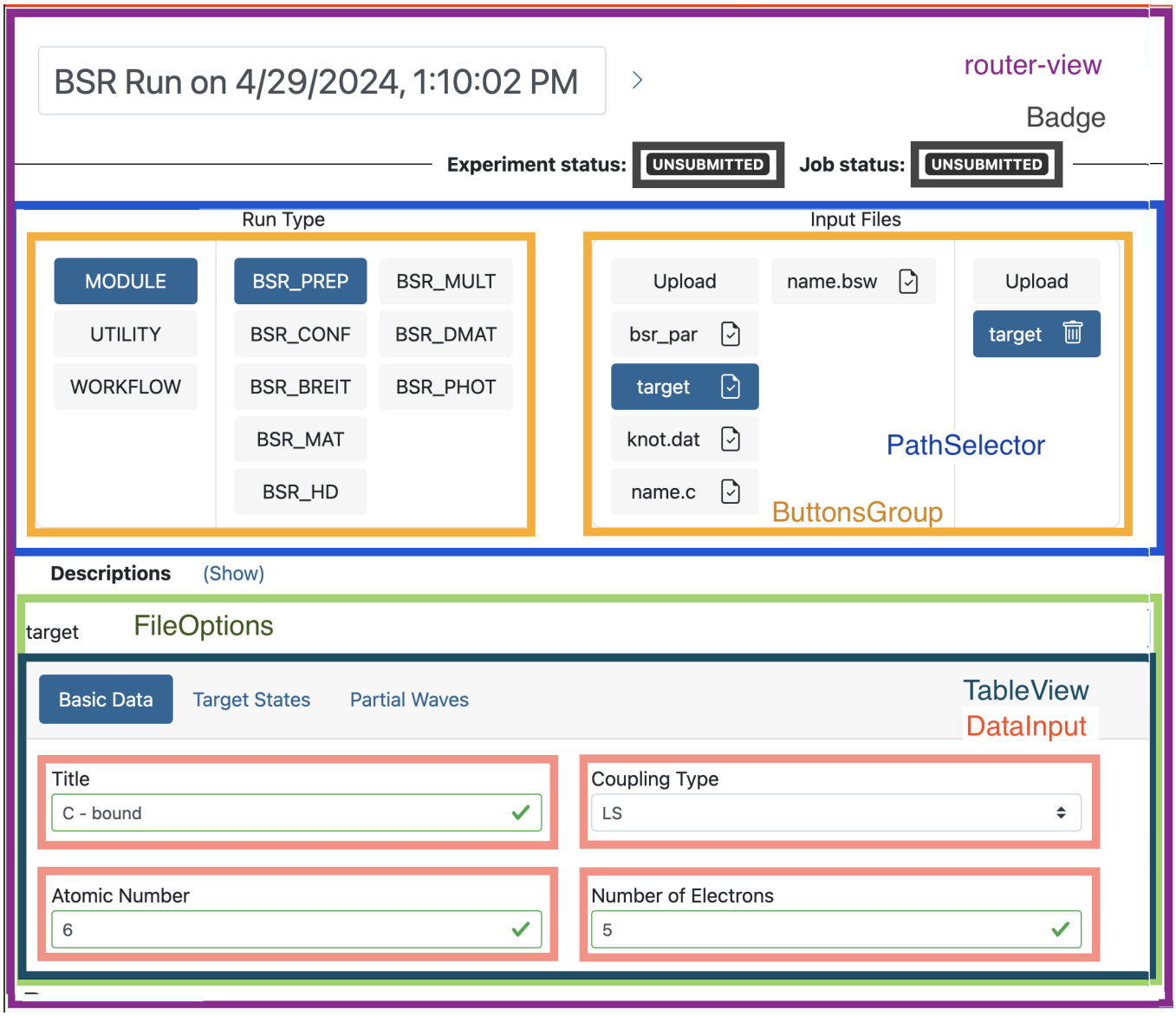

FIG. 5. "Create new run" interface with each component marked and labeled. The purple box is the `router-view`, which updates based on the selected page, in this case the "Create new run" page. `Badges` are included (black boxes) to display the Experiment and Job statuses. The blue box is the path selector, which contains two `buttonsGroups` (orange boxes) for selecting the BSR run type and the input files. The green `FileOptions` box allows the user to select a text, table, or a plot view for a selected file. A table view is shown in the deep green box. It contains multiple `DataInput` components (red boxes) which allow for the selection or setting an input value for the selected input file in the Input Files `buttonsGroup`.

emitted into the event bus. This is picked up and displayed through vue-toastification [18], as shown in Fig. 8. This lets the user know when something goes wrong, so they can try again later or potentially resolve the issue.

Loading screens were handled through the Vuex store and the `LoadingOverlay` component. The store allowed for similar flexibility to the event bus, while also being able to mantain its state between different instances of the component. Furthermore, in order to help communicate that things are happening to the user, the loading overlay component has a spinner to indicate this and a label on the loading screen to convey the current process.

C. Plotting

The BSR interface uses largely the same code for plotting as tRecX, albeit with some tRecX-specific functionality replaced. On the back end, a call to the plotting API would take in plot parameters along with a list of input files to plot. The code used to generate the graph was taken directly from tRecX, enabled by the fact that the majority of the code was generalized to work outside of just tRecX's plottable data files. The plot parameters contain the columns to be graphed, the axes to be plotted along, and the flags to change

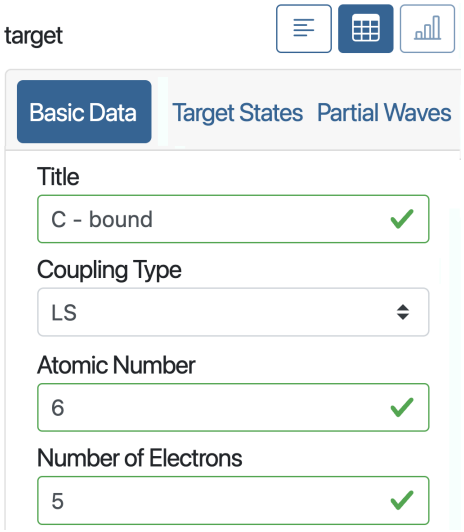

FIG. 6. A table view for target using the key value pair format.

Listing 4. Example on incorporating error handling and loading screens.

```
this.$store.commit("loading/START", {
  key: "plot",
  message: "Fetching Plot Parameters"
});

try {
  this.plotParameters = await this.$store.
      dispatch(
    "plotParameters/fetchPlotParameters"
  );
} catch (error) {
  eventBus.$emit("error", {
    name: "Error while trying to load plot
        parameters",
    error
  });
}

this.$store.commit("loading/STOP", {
  key: "plot",
  message: "Fetching Plot Parameters"
});
```

the plotting utility's behavior.

On the front end, the returned base64-encoded image is displayed through the `FileOptions` component in a graph view. Figure 9 shows an example of this. When graphing the user can select an x-axis, multiple y-axes, flags, and multiple runs to be plotted. A list of the available flags and their descriptions is provided via a popup modal.

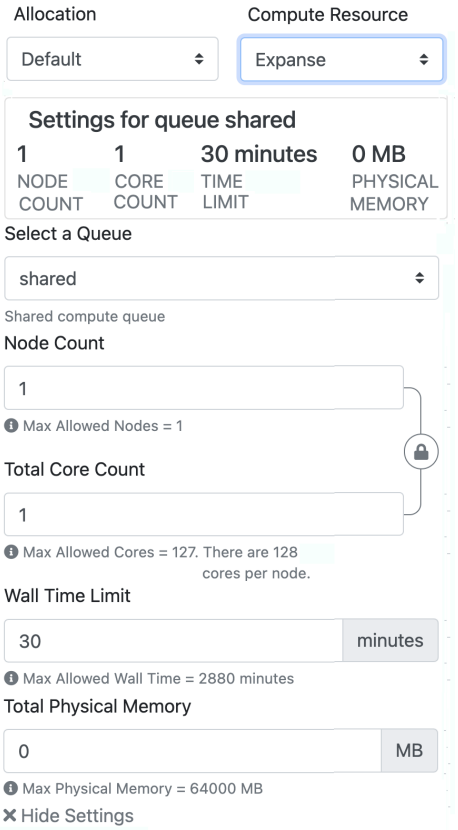

FIG. 7. Interface for selecting compute resources. Users can select the Compute Resource from a list of machines that the application has already been deployed on, and which allocation to be charged against. Users can also change the queue the job is submitted to, the node / core count, the wall time limit, and the total physical memory required.

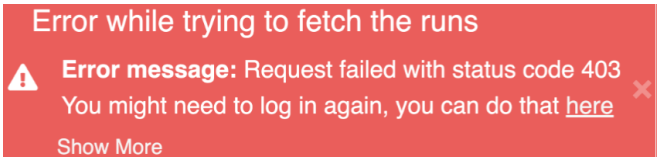

FIG. 8. An example error message as displayed on the AMOSGateway. The error is displayed in the top-left corner of the screen and gives hints on how the user can correct it.

V. BSR WORKFLOWS

As described in Section II, BSR calculations are typically performed by executing a series of modules and utilities in a prescribed order with the goal of obtaining one specific observable. The BSR suite, therefore, lends itself to the creation of workflows, which group these modules, utilities, and relevant calculation parameters in the correct order of execution. Traditionally housed in simple shell scripts and passed amongst users, the AMOS Gateway is an attractive alternative to implement BSR workflows, offering many advantages, particularly for novice users. Two types of BSR

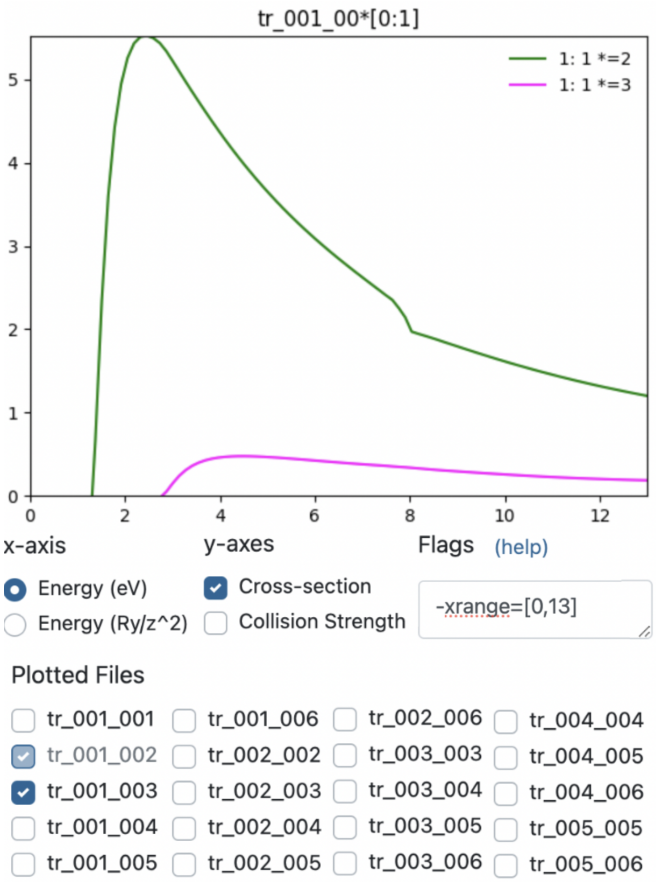

FIG. 9. Plotting view of two cross sections for electron-carbon collisions in units of a_0^2. The green line on top shows the results extracted from the *tr_001_002* file (excitation of the first excited state from the ground state), while the cyan line on the bottom is from the *tr_001_003* file (excitation of the second excited state). The legend uses the $*$ wildcard from the title of the graph.

workflows are currently supported by the AMOS Gateway: ready-made workflows that are geared towards novice users, and chained task workflows, which allow advanced users the flexibility to create their own workflows.

A. Ready-made workflows

For novice users, executing the many different steps of a BSR calculation with their many inputs can be an intimidating and difficult task. Novice users also typically want to perform small, quick calculations, and are more interested in examining the final output of a BSR calculation rather than a thorough investigation of the suites numerical and physical capabilities. For these users, it is much more convenient to run BSR calculations in the form of a workflow: a pre-arranged sequence of BSR executables that produce one final output that is of interest to the user. These workflows, coupled with the provision of sets of small example calculations made available on the AMOSGateway, are helpful in engaging novice users and allowing them

Name	Executables	Inputs	Output
Bound	`bsr_prep`	*target*	
	`bsr_conf`	*name.c*	
	`bsr_breit`	*name.bsw*	
	`bsr_mat`	*knot.dat*	
	`bsr_hd`	*bsr_par*	
	bound_tab		*bound_tab*
Scattering	`bsr_prep`	*target*	
	`bsr_conf`	*name.c*	
	`bsr_breit`	*name.bsw*	
	`bsr_mat`	*knot.dat*	
	`bsr_hd`	*bsr_par*	
	`sum_hh`		*H.DAT*
STGF	`stgf`	*target*	
	`add_stgf`	*H.DAT*	
	`sec_om`	*dstgf*	*tr_nnn_nnn*

TABLE III. Description of the completed AMOS Gateway BSR workflows. Shown are the workflow names, the ordered list of BSR modules and utilities that constitute the workflow, the input files required from the user, and the main output file. The file *bound_tab* contains an ordered list of the bound-state energies for the system under investigation, *H.DAT* contains the diagonalized Hamiltonian matrix for all the partial waves, and *tr_nnn_nnn* contains the electron-impact excitation cross section for an individual transition.

to become familiar with the BSR suite. Novice users could include undergraduate and graduate researchers, and scientists from domains that utilize AMO data.

There are currently three workflows on the interface ready for use: bound, scattering, and STGF. The bound and scattering workflows take in the same input files, which are prompted by the interface: *target*, *knot.dat*, *bsr_par*, *.c* and *.bsw* files. The bound workflow calculates bound-state energies, the scattering workflow calculates the *H.DAT* file containing the diagonalized Hamiltonian matrix, and the `STGF` workflow uses this *H.DAT* file along with the target file to produce scattering cross sections. The BSR suite executables, input files, and major outputs are summarized in Table III. Users additionally specify the initial and final partial-wave indices to determine the range of partial waves for which bound states and the *H.DAT* file are calculated in the bound and scattering workflows, and can upload a file containing desired numerical parameters for the `STGF` workflow to consume. Additional workflows to calculate, for example, dipole polarizabilities, oscillator strengths, and photoionization cross sections (see Fig. 1) could also be produced.

We hope that these workflows, along with restricted parameter choices and the provision of sample inputs for each workflow type, will make calculations by novice users much more achievable. While using a workflow, users are still able to make the usual range of selections regarding computing resources.

B. Chained Task Workflows

While the ready-made workflows offer novice users a convenient way to perform BSR calculations, they, by design, considerably limit the extent to which a user can customize a calculation. Ready-made workflows are therefore less useful for advanced users, who may wish to perform adjustments to the calculation after every individual step, or wish to perform very large calculations that could not be completed within the 48-hour walltime limit imposed by most supercomputing centers. However, manually handling the upload / download of input and output data from a multi-step calculation can be time-consuming, and so the AMOSGateway includes options for chained task workflows to make calculations more convenient for advanced users.

An advantage of the AMOSGateway is that chaining files in the interface is fairly simple. The first run in a multi-step calculation can be executed in the usual manner, with the user uploading input files or selecting them from example suites on the Gateway. Once the first part of the calculation has finished, the `Create new run with outputs` button creates a new run and automatically populates the inputs with all the files from the last run. This avoids the user having to download and re-upload input/output files.

Behind the scenes, if data from a previously executed task (run) are needed by subsequent tasks, the gateway can handle the staging of data on the remote HPC resource. This can be assumed and no data upload will be needed. However, it requires the handling of the previous run, such as the `JobID` to enable the data availability. Chaining runs this way allows faster execution, as the uploading step is bypassed, and gives users greater flexibility in how they perform calculations.

VI. USING THE INTERFACE

The interface is composed of multiple pages: the home page, the runs page, the views page, the tutorials page, and the create/edit/view run page. Across all the pages is a navigation bar for quick access.

A good place to start when learning the interface is the tutorials page. Each tutorial has a description which will explain what the run carries out and which inputs could be modified with interesting results.

The first page on the interface is the home page. This page contains a description of the BSR codes, a walk-through for using the interface, and quick access to the most recently updated runs. Additionally, the description provides a link to [8], which gives the most comprehensive documentation of the BSR codes currently available.

The runs page displays all the runs in a table, with links to each run, showing their job status and date of the last update, in addition to providing quick access `buttons`. These `buttons` allow cloning, submitting, or deleting of the run. In order to find runs, the page has a search bar and allows the user to sort by name, status, or date modified.

Runs can also be selected in the table. This will pop up new `buttons` to allow the user to delete all the selected runs or save them as a **view**.

If any of these runs contain plottable files, a sidebar will appear where these files can be compared with each other as shown in Fig. 10. While there is only one kind of plottable file right now, the first selection box allows the user to choose which file to plot. The plot parameters selection box makes it possible to view and choose from the previously used plotting parameters. One can then select the x- and y-axes to be plotted and add any additional flags to change the plotting. The descriptions are available in a pop-up modal.

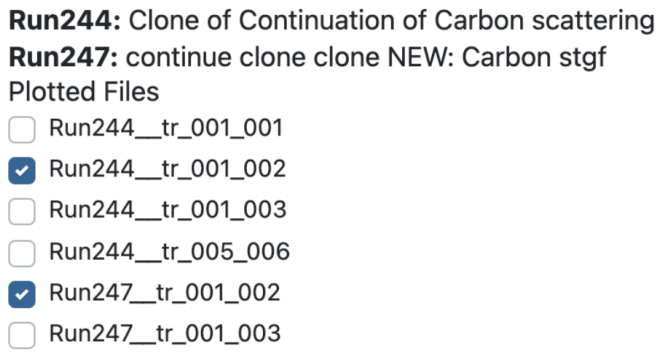

FIG. 10. Plotting sidebar on the runs page. A user-generated description of each run number is displayed (default descriptions are also generated), and files selected for plotting are highlighted with a check mark.

VII. WALK-THROUGH OF SPECIFIC EXAMPLES

The goal of these walk-throughs is to start from the home page and guide the user through adding files, editing files, saving, submitting, checking the status, viewing the outputs, and chaining computations for specific examples. These example runs are provided on the tutorials page, which can be cloned. Alternatively, one can open up the interface and follow one of these examples to get a grasp for using the interface. In addition to the text walk-throughs, videos demonstrating a bound-state calculation and a scattering cross-section calculation are available on YouTube.

First, go to https://amosgateway.org and log in or create an account. It is recommended that an institutional login is used so that their email is pre-verified and secure authentication is established. The users need to be authorized to access the applications and resources by the gateway administrator. Once logged in and authorized, click on the dropdown in the navigation at the top of the page labeled "**Workspace**" and select "**BSR Django App**". This will redirect the users to the BSR interface page from the default Workspace dashboard.

A. Carbon bound-state calculation

On the BSR interface page, click on "**Create new run**" on the right side of the screen. This leads to the new run page, where you can change the default name for the run by clicking on the title at the top. Additionally, if you click the drop-down arrow next to the title, you will be able to add a description, which can provide additional metadata for the run. To select the type of run, scroll down to the box labeled "**Run Type**" and select "WORKFLOW" on the left side. "BOUND" module is selected by default on the right side of the **Run Type** box. Now, since "Upload" is selected in the **Input Files** box, you can scroll down to the Uploading section and click "**Upload from storage**", which will open up the gateway's user storage. The files for this example computation are available in shared/Sample_Inputs/BSR/Carbon_bound_2024. Navigate into the folder through the file system and click the selection box at the left of the header row. This selects all the files, and clicking "**Open**" at the bottom right will upload them to for the run. Looking at the files listed in the "**Input Files**" box, notice that the file icons now have checkmarks in them, thereby indicating that the files have been uploaded.

If you want to edit or view any of these input files, clicking on a file in **Input Files** will open it up in file viewer below. Depending on the file type, it might support being viewed in different ways. Directly above the file viewer and to the right are three icons representing a text, table, and plot view, respectively. If the selected file does not support a view, its button will be disabled. For this run, only *bsr_par*, *target*, and *knot.dat* are meant to be edited by users. Therefore, they are the only ones with a table view. In addition to the input files, the bound-state calculation requires the parameters "Index of the first partial wave" and "Index of the final partial wave" to be set. These will define the range of partial waves for which bound states will be calculated. Since there are four partial waves in this calculation, try setting the first index to 1 and the final index to 4.

In addition to the inputs to the run, there are compute resource settings that can be set. To change the defaults, click on the box containing all the settings and adjust each field to suit the run. The defaults should work for the current run. If you want a short run to complete more rapidly, change "Select a Queue" to "shared" in order to have the run put in the shared queue.

Now the run can be saved and submitted using the `buttons` at the bottom of the page. Once submitted, the user is redirected to the Runs page and the status of the run can be seen near the top of the run page. The runs are also listed on the left navigation bar of the page and are accessible through the "Runs" link. Wait for the run to be completed. This calculation will typically take 3-5 minutes, depending on the waiting time at the compute resource. Once the run is **completed** as suggested by the Status, you will be able to see the input files, intermediate files, logs, job info, and *bound_tab*. Click on *bound_tab* to view its contents and you will see the excited states of carbon. To double check you got the correct outputs, the first entry "`2p_3s_3P`" should have an `E_au` value of `-37.48158374`.

B. Carbon scattering cross-section calculation

Moving on to a slightly more complex calculation, the scattering workflow starts out the exact same way as above. From home go to create a new run, change the title and description, and select " WORKFLOW" followed by " SCATTERING" under "**Run Type**". From here click "Upload from storage", navigate into the folder at shared/Sample_Inputs/BSR/Carbon_LS_2024, select all of the files using the checkbox in the header row, and click "**Open**".

The partial-wave index parameters can be changed here as well to include any of the 12 partial waves provided. It is important to note for these computations that the range should not only include one partial wave such as 1-1, because this will cause the first cross-sections to be trivial. For this example computation, set the range to 1-12.

The default compute settings should work here as well, but you can change the queue to "Shared" in order to potentially get the computation to start sooner. You can then submit the run and wait for it to finish, which should take 4-7 minutes, depending on the waiting time at the compute resource. Once the run is completed, there will be a button in the top right corner of the page labeled "**Create new run with outputs**". This will create a new run and upload all the output files from the previous run. Once you are on this new page, select "WORKFLOW" followed by "STGF". Then you will need to upload shared/BSR/Carbon_LS/dstgf specifically for the `STGF` workflow. Do this by navigating to the folder, finding *dstgf* from the list of files, selecting it, and clicking "Open".

The parameters for this computation need to be filled out. A good input for this is to set both the initial and final scattering-state indices to 1 and 3. This will set the range for the initial indices to be 1-3 and the range for the final indices to be 1-3. From here, set the queue to "Shared" again and then submit the run. This run should take about 3-5 minutes. After it is completed, there will be a plottables section available to select. Inside this are the electron scattering cross sections produced by the computation with labels *tr_001_001*, *tr_001_002*, *tr_001_003*, etc.

These files can plotted and viewed as text. You can switch between the two using the icons directly above the file viewer. Using Plot View for the file, you can select the x- and y-axes to be plotted. This will produce the plot as an image. There are some additional flags that can be added to change the plotting. To see a list of the flags, click "(help)" above the flags box. Specifically, the cross section can be seen by plotting sigma vs. eV. More *tr_nnn_nnn* files from the same run can be plotted alongside the current file by selecting more files under "Plotted Files".

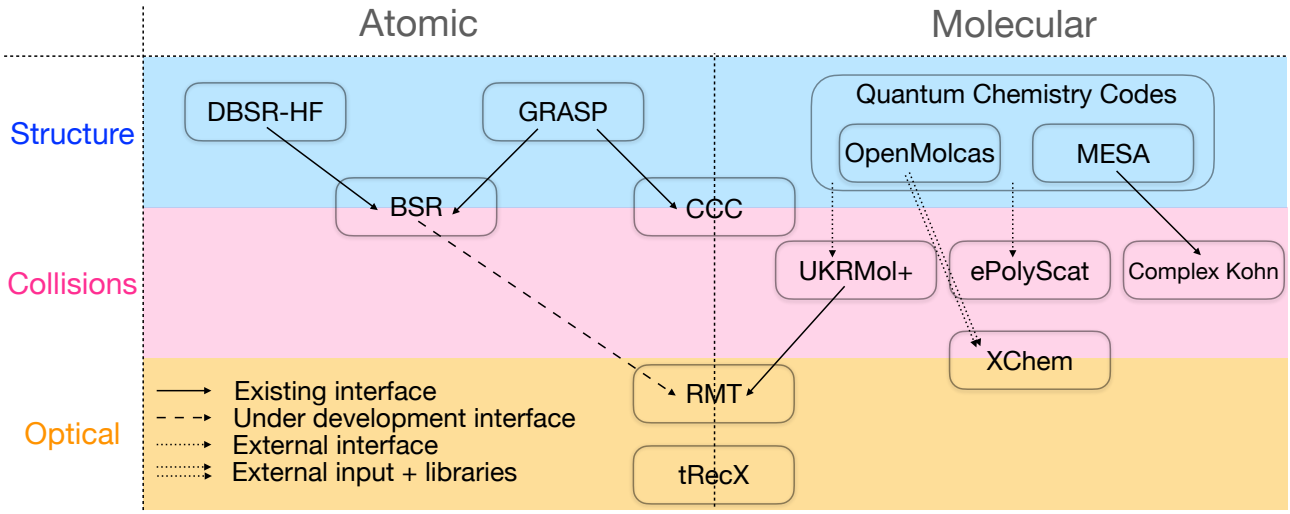

FIG. 11. Diagram of relationships between software suites hosted on the AMOSGateway. Existing relationships / interfaces are denoted with solid black lines, relationships which require external software suites or libraries to generate inputs are denoted with a dotted line, and interfaces which are currently under development are marked with a dashed line. The AMOSGateway developers plan to extend the Chained Task Workflow functionality to allows users to perform calculations using multiple different software suites.

VIII. COMPUTATIONAL RESOURCES

The availability of appropriate computational resources to perform many of the calculations on the AMOS gateway is critical to its success. As part of the operation of the gateway, community allocations are available on a number of the computers funded by the National Science Foundation via the ACCESS program and on the leadership class machines at the Texas Advanced Computer Center. These include Expanse, Bridges, Delta, Anvil, Ookami, and Frontera. The gateway developers write proposals for these allocations, which are awarded yearly on a competitive basis. Individual allocations can be registered to use the gateway as well. When a person is authorized as a gateway user, they have access to these allocation on all the machines hosted by the AMOS gateway. While the gateway has allocations on many NSF machines, executables for some of the codes may only be available on a subset of the machines. For example, some codes are GPU-capable, and they are available on GPU resources. It is also possible for a group, having its own allocation, to make all or part of that allocation available for use on the gateway while restricting access to only authorized group users. This allows group members to freely use the codes and computational resources available on the gateway.

IX. CONCLUSIONS AND OUTLOOK

In this paper, we have introduced the AMOS gateway using the BSR method for atomic bound-state and electron-scattering calculations as an example. The gateway hosts a number of other software suites to study molecular collisions, photoionization and photodetachment, and the interaction of short, intense radiation with atoms and molecules. These include the Convergent Close-Coupling method, the UK R-matrix suite, the time-dependent R-matrix code, XCHEM and ePolyscat for molecular photoionization, and MESA, a code to compute electron-molecule collisions and photoionization using the complex-Kohn approach. Figure 11 shows an overview of the current relationships between the various codes. The AMOSGateway developers plan to extend the Chained Task Workflow functionality to allows users to perform calculations using multiple different software suites. The AMOS science gateway provides a portal of tools that, in principle, enables both novice and expert users to perform calculations at quite sophisticated levels. Importantly, it can also be used to teach students the principles that underlie a given scientific field, e.g., atomic, molecular, and optical physics and chemistry.

We hope that this democratization of science will lead to a rapid advancement of the field and expand the user base far beyond the local developers who are typically responsible for the development of the application. The gateway makes the software accessible to everyone, especially when user-friendly GUIs are part of the gateway landscape. We encourage readers who have applications that could benefit from our developments to contact the authors. Expanding the code base to include other areas of computational AMOS is certainly desirable and part of our mission.

X. ACKNOWLEDGEMENTS

This work is currently supported by the National Science Foundation (NSF) under the Cyberinfrastructure for Sustained Scientific Innovation (CSSI) Frameworks Project OAC-2311928: *An Advanced Cyberinfrastructure for Atomic,*

Molecular, and Optical Science (AMOS): Democratizing AMOS for Research and Education. Funding for initial workshops exploring the construction of the AMOS gateway was provided by NIST and the MolSSI. In addition, the project benefited enormously from the NSF-supported projects *Science and Engineering Discovery Environment* (XSEDE) and its successor *Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support* (ACCESS) under allocation PHY200012: AMP: A Science Gateway for Atomic and Molecular Physics. Computational resources were also provided by the Texas Advanced Computing Center (TACC) under the Frontera Pathways allocation PHY23020, and through an Institute for Advanced Computational Science Director's Discretion allocation on Ookami at Stony Brook. T. W. would like to acknowledge support from the NIST Summer Undergraduate Research Fellowship (SURF) and Professional Research Experience Program (PREP) programs.

[1] B. I. Schneider, K. Bartschat, O. Zatsarinny, I. Bray, A. Scrinzi, F. Martin, M. Klinker, J. Tennyson, J. D. Gorfinkiel, and S. Pamidighantam, "Atomic, molecular and optical sciences gateway," (2019).
[2] B. I. Schneider, "AMO for All: How Online Portals Are Democratizing the Field of Atomic, Molecular and Optical Physics," (NIST blog, 2022).
[3] B. I. Schneider, K. Bartschat, O. Zatsarinny, I. Bray, A. Scrinzi, F. Martín, M. Klinker, J. Tennyson, J. D. Gorfinkiel, and S. Pamidighantam, arXiv preprint arxiv:2001.02286 (2020).
[4] K. R. Hamilton, K. Bartschat, N. Douguet, S. V. Pamidighantam, and B. I. Schneider, Computing in Science & Engineering **25**, 68 (2023).
[5] P. G. Burke, *R-Matrix Theory of Atomic Collisions*, Springer Series on Atomic, Optical, and Plasma Physics, Vol. 61 (Springer-Verlag, 2011).
[6] K. A. Berrington, W. B. Eissner, and P. H. Norrington, Comp. Phys. Commun. **92**, 290 (1995).
[7] O. Zatsarinny and C. F. Fischer, J. Phys. B: At. Mol. Opt. Phys. **33**, 313 (2000).
[8]O. Zatsarinny, Comp. Phys. Commun. **174**, 273 (2006).
[9] O. Zatsarinny and K. Bartschat, J. Phys. B: At. Mol. Opt. Phys. **46**, 112001 (2013).
[10]O. Zatsarinny, "BSR Github page," (2024).
[11] S. Pamidighantam, D. D. Silva, M. Christie, B. Schneider, A. Scrinzi, and S. Mhatre, Practice and Experience in Advanced Research Computing (PEARC '23) , 237 (2023).
[12] M. Christie, S. Marru, E. Abeysinghe, D. Upeksha, S. Pamidighantam, S. P. Adithela, E. Mathulla, A. Bisht, S. Rastogi, and M. Pierce, Practice and Experience in Advanced Research Computing (PEARC '20) , 160 (2020).
[13]Vue, "The Progressive Javascript Framework," (2024).
[14]BootstrapVue, "Javascript Frontend Toolkit v4," (2024).
[15] Vuex, "The state management pattern + library for vue.js applications," (2024).
[16] AXIOS, "Promise-based HTTP client for the browser and node.js," (2024).
[17] Vue Router, "Expressive, configurable and convenient routing for vue.js," (2024).
[18] Vue Toastification (for Vue 2), "Customized notification in vuejs applications," (2024).