Open Chemistry, JupyterLab, REST, and Quantum Chemistry

Marcus D. Hanwell[1,2], Chris Harris[1], Alessandro Genova[1], Mojtaba Haghighatlari[3], Muammar El Khatib[4], Patrick Avery[1], Johannes Hachmann[3], and Wibe Albert de Jong[4]

[1]Kitware
[2]Brookhaven National Laboratory
[3]University at Buffalo – State University of New York
[4]Lawrence Berkeley National Laboratory

August 26, 2020

Abstract

Quantum chemistry must evolve if it wants to fully leverage the benefits of the internet age, where the world wide web offers a vast tapestry of tools that enable users to communicate and interact with complex data at the speed and convenience of a button press. The Open Chemistry project has developed an open source framework that offers an end-to-end solution for producing, sharing, and visualizing quantum chemical data interactively on the web using an array of modern tools and approaches. These tools build on some of the best open source community projects such as Jupyter for interactive online notebooks, coupled with 3D accelerated visualization, state-of-the-art computational chemistry codes including NWChem and Psi4 and emerging machine learning and data mining tools such as ChemML and ANI. They offer flexible formats to import and export data, along with approaches to compare computational and experimental data.

Keywords: reproducibility, computational chemistry, open infrastructure, automation, visualization

Introduction

We live in a golden age where data can be transmitted at light speed to almost anywhere on the planet. Yet modern scientific publishing, and indeed much of the research process, languishes in a time where printing presses rule. It is long past time that these processes and publication practices were modernized - there is no need to primarily rely on PDF tables and static images in an age where advanced JavaScript applications can run on cell phone browsers using hardware accelerated rendering in 2D and 3D and where augmented and virtual reality is emerging as a powerful, immersive alternative to more traditional approaches.

It is time for a paradigm shift, that embraces fully digital workflows where all steps of research projects are documented, one in which modern tools can be used to "show all steps of our research process". To that end the authors of this paper have been developing an ambitious open source platform to realize the dream of open science in a fully connected digital age. PDFs and flat images are relics of the past, attempts to map the limitations of the printing press to the digital world. It is time to push the scientific research enterprise forward, and drive innovation in software to a point where interactive data outputs can be published for the whole world to see. We must look at the technologies available in modern web browsers to offer seamless publication of data, code, and written results by leveraging the best open source software. Extending this software where necessary to offer specialized views, entry points, and direct access to raw, partly processed data, to tools that extract the key information from the data, and even notes, linked from figures in published works.

The modern scientific workflow takes place increasingly on a computer, or distributed set of computers/nodes/images. This needs to be reflected in, and embraced throughout the dissemination of research. Computational chemistry by its very nature is a prime example of this, and we have the tools to document all steps in producing, processing/analyzing data, visualizing data and linking that with the final publication. More and more research is multidisciplinary in nature, with experimentalists, theoreticians and computational scientists working closely together in their pursuit of scientific discoveries that advance society. Integration of diverse data resources and analysis tools into complex research workflows can be made much more accessible with modern technologies. In this paper we will discuss our approach to the handling of data and metadata, storing them on a flexible data server platform, representing rich data that can be exported, integrating computational chemistry data generators (quantum chemistry and machine learning codes) with experimental data, and approaches to generate and access the data through the web using browser-based and desktop-based tools.

The project described has been developed in the open on GitHub using OSI-approved open source licenses (*Licenses & Standards — Open Source Initiative*, n.d.), primarily the 3-clause BSD license (*The 3-Clause BSD License — Open Source Initiative*, n.d.). Issues were used to track new features to be developed, and bugs in existing functionality. Pull requests offer a number of commits in a logical branch to add new features of fix bugs, where possible these feature automated testing and line-by-line review of code additions/removals using the web interface. Focused software repositories were used for each component, such as the server code in Python (*Welcome to Python.org*, n.d.), the web client code in JavaScript/TypeScript, and the deployment code/image build instructions. A coordinating repository was also used to discuss high level development of the platform within the "OpenChemistry" (*Open Chemistry*, n.d.) organization on GitHub.

The GitHub platform offers access control for registered users, including individual and team level access control. Integrations with a number of other platforms create a rich ecosystem where continuous integration jobs can be run for pull requests, when the main development branches are merged, and when release tags are created. Commits and release tags can be cryptographically signed, and releases offer snapshots of the software at a particular point in time. Pull requests can be made by anyone, code can be reviewed by anyone, but only approved developers can merge code to the main development branches. This offers a level of transparency and accessibility that is often the norm in open source software development, but is not so common when developing code for scientific research. All code developed, along with all previous versions of the code, remain available using these standard technologies and platforms. More recently this has been used to document the data/metadata formats, the build process for images, and the deployment of services.

Handling Data and Metadata

If things work well the data and metadata standards used will rarely be seen by end users, but they will enable the storage of data within server instances, and the seamless transfer of data into or out of these systems. Flexible standards must have formats that can be used from a number of programming languages ranging from compiled languages such as C, C++, and Fortran on supercomputers/high performance computing resources to perform quantum chemistry calculations through to interpreted languages such as Python for data analysis and JavaScript/TypeScript in web frontends or C/C++ in desktop applications. They should also be suited to the needs of persistent data publication independent of any particular database technology or programming language.

These considerations led to the choice to make use of JavaScript Object Notation (JSON) (*JSON*, n.d.) as a core standard for data and metadata, with a view to using related technologies such as JSON-LD (JSON for Linked Data) (*JSON-LD - JSON for Linking Data*, n.d.) where appropriate. Large data is preferably stored in binary formats, which is where HDF5 (*HDF Group - HDF5*, n.d.) was seen as one strong contender and more recently MessagePack (*MessagePack: It's like JSON. but fast and small.*, n.d.) has gained traction due to its JSON-like structure and wide language support thanks to its simple binary specification. JSON also lends itself to use in BSON (*BSON (Binary JSON) Serialization*, n.d.) and jsonb (*PostgreSQL:*

Documentation: 9.4: JSON Types, n.d.) - two binary JSON specifications used in MongoDB (*The most popular database for modern apps*, n.d.) and PostgresSQL (*PostgreSQL: The world's most advanced open source database*, n.d.) respectively.

In order to effectively share chemical data we must establish data and metadata standards capable of representing everything we wish to communicate. Further, it must offer routes to extending the standards without causing breakage and churn in existing data. Ideally communities should form to establish best practices, and propagate this to a number of codes to prove viability and offer a body of work that demonstrates the advantages of the approaches shown. A number of existing formats have been used such as XYZ (*XYZ file format - Wikipedia*, n.d.), SDF, XML-based formats such as CML (Phadungsukanan et al., 2012; Murray-Rust & Rzepa, 2011; Murray-Rust et al., 2011; de et al., 2013) and more recently JSON-based formats such as Chemical JSON (Hanwell et al., 2017). Open Babel (O'Boyle et al., 2011), RDKit (*RDKit: Open-Source Cheminformatics Software*, n.d.), cclib (O'boyle et al., 2008) and ASE (Hjorth et al., 2017) offer conversion between these and many other formats, aiding in the normalization of data from different sources when ingesting data.

The JSON specification was chosen for its wide language support, with fast simple parsers available in every language considered including those outlined previously: Python, C, C++, Fortran, JavaScript and TypeScript. It is not a part of the core language, but it has become established as a simple container format that can easily map to data structures in all of these languages, for example Python dictionaries, C++ maps, and with a little work the array and some shape information easily maps to NumPy arrays (van der Walt et al., 2011) in Python or one of a number of structures in C++ such as the Eigen (*Eigen*, n.d.) matrix structures or Vector3 for position information. The text format is easy to inspect for casual users, and there are clear routes to migrate to very similar binary containers in the future as outlined earlier. Due to the simplicity of the container format fast parsers can be used, and the loss of precision is of a secondary concern for visualization. It will of course be very important if exchanging data between quantum chemistry codes, and would warrant further consideration there.

The block below shows the Chemical JSON (*OpenChemistry/chemicaljson*, n.d.) representation of a water molecule. It contains a unique key of "chemicalJson" that can be used to identify the convention being used, along with a numeric version identifier of '1'. The first iteration used "chemical json" with a version of "0" before the decision was taken to switch to camelCase to simplify client code. The "atoms" object contains "elements" identifying the atoms by their atomic number. The "coords" object contains a "3d" array that has 3N numbers representing the Cartesian coordinates in Angstroms. There is also a "bonds" object containing connectivity of atoms (using a zero-based index), and the bond orders.

```
{
  "chemicalJson": 1,
  "atoms": {
    "elements": { "number": [ 8, 1, 1 ] },
    "coords": {
      "3d": [ 0.0000,  0.0000, -0.0685,
              0.0000, -0.7890,  0.5437,
              0.0000,  0.7890,  0.5437 ]
    }
  },
  "bonds": {
    "connections": {
      "index": [ 0, 1,
                 0, 2 ]
    },
    "order": [ 1, 1 ]
  }
}
```

The equivalent molecule can be seen using the QC JSON Schema (*MolSSI/QCSchema*, n.d.), where the molecule geometry is expressed in Bohr, and the atom type uses the string symbols rather than atomic numbers. The QC JSON Schema is at an earlier stage of development, with some pull requests under review adding support for bonding, basis sets, and more. It also has a fairly comprehensive set of possibly calculated properties expressing the numeric results of calculations such as the energy, number of iterations, etc. The authors have been collaborating with MolSSI developers to develop the QC JSON Schema, and help drive its standardization in the hopes of wider adoption in the community. Recent work has added support for basis sets into the standard.

```
{
  "schema_name": "qc_schema_output",
  "schema_version": 1,
  "molecule": {
    "geometry": [ 0.0,  0.0,    -0.1294,
                  0.0, -1.4941,  1.0274,
                  0.0,  1.4941,  1.0274],
    "symbols": ["O", "H", "H"],
    "connectivity": [ [0, 1, 1], [0, 2, 1] ]
  }
}
```

The Chemical JSON schema has been in development and use since 2011, focusing on the needs of the Avogadro chemical editor(Hanwell et al., 2012) and the associated Open Chemistry open source projects. It has as such been developed to enable the visualization and analysis of chemical systems, supporting elements such as the basis set, molecular orbitals and other electronic structure calculation outputs of the system to offer a flexible format. This has been extended(Hanwell et al., 2017) to support multiple steps in a reaction coordinate calculation for example (one basis set, multiple 3D geometries and molecular orbital matrices). There is also support for vibrational modes, and 3D cubes representing calculated molecular orbitals. An extended example for water, available in the Chemical JSON repository, shows a basis set, molecular orbitals and related data from a Psi4(Parrish et al., 2017) energy calculation.

```
{
  "chemicalJson": 1,
  "atoms": {
    "coords": {
      "3d": [ 0.000, 0.0, -0.058934372265,
             -0.811, 0.0,  0.467665627735,
              0.811, 0.0,  0.467665627735 ]
    },
    "elements": { "number": [ 8, 1, 1 ] },
  },
  "basisSet": {
    "coefficients": [ 0.0018311, 0.0139501, 0.0684451, 0.2327143, 0.470193, 0.3585209, -0.1107775, -0.1
    "exponents": [ 5484.6717, 825.23495, 188.04696, 52.9645,
                   16.89757, 5.7996353, 15.539616, 3.5999336,
                   1.0137618, 15.539616, 3.5999336, 1.0137618,
                   0.2700058, 0.2700058, 18.731137, 2.8253937,
                   0.6401217, 0.1612778, 18.731137, 2.8253937,
                   0.6401217, 0.1612778 ],
    "primitivesPerShell": [ 6, 3, 3, 1, 1, 3, 1, 3, 1 ],
    "shellToAtomMap": [ 0, 0, 0, 0, 0, 1, 1, 2, 2 ],
    "shellTypes": [ 0, 0, 1, 0, 1, 0, 0, 0, 0 ]
  },
  "inputParameters": {
    "basis": "6-31g",
    "functional": "b3lyp",
    "task": "energy",
    "theory": "dft"
  },
  "orbitals": {
    "electronCount": 10,
    "energies": [ -520.4913730891953, -27.318216947706304,
                  -14.7250600604168, -9.188141853212901,
                  -7.8296406659767, 1.51597347252055,
                  4.01250920531785, 22.885155786440702,
                  23.307394848261552, 24.2824604087582,
                  25.337881189307502, 29.451154153465502,
                  37.9528616074071 ],
    "moCoefficients": [ 0.9950345, 0.0278763, 0.0, 0.0,
                        0.0020759, -0.0130155, 0.0, 0.0,
                        -0.0023909, 2.28e-05, 0.0030608,
                        2.28e-05, 0.0030608, 0.2110473,
                        -0.4670282, 0.0, 0.0, -0.1390958,
                        -0.4565485, 0.0, 0.0, -0.0652543,
                        -0.1474718, -0.0010925, -0.1474718,
                        -0.0010925, 0.0, 0.0, 0.5114143, 0.0, 0.0, 0.0, 0.2288377, 0.0, 0.0,
                        -0.2680075, -0.1416302, 0.2680075,
                        0.1416302, 0.0831513, -0.1855981, 0.0, 0.0, 0.5583771,
                        -0.3437312, 0.0, 0.0, 0.4007392, 0.1300784, 0.1003324,
                        0.1300784, 0.1003324, 0.0, 0.0, 0.0, 0.6447118, 0.0,
                        0.0, 0.0, 0.5066596, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0923385, -0.1620552, 0.0, 0.0, -0.2736157,
                        -1.1033256, 0.0, 0.0, -0.4115525, 0.1142114,
                        0.9370367, 0.1142114, 0.9370367, 0.0, 0.0, 0.4162705,
                        0.0, 0.0, 0.0, 0.7736278, 0.0, 0.0, 0.0843618,
                        1.2638376, -0.0843618, -1.2638376, 0.0508968,
                        -0.2289781, 0.0, 0.0, -0.5061012, 0.1767856, 0.0, 0.0,
                        0.0962835, 0.9030719, -0.6154998, 0.9030719,
                        -0.6154998, 0.0, 0.0, 0.3090174, 0.0, 0.0, 0.0,
                        0.4348612, 0.0, 0.0, 0.9900298, -0.6814988,
                        -0.9900298, 0.6814988, 0.0, 0.0, 0.0, 0.9593712, 0.0,
                        0.0, 0.0, -1.038914, 0.0, 0.0, 0.0, 0.0, 0.0,
                        0.0557295, -0.2870743, 0.0, 0.0, 0.8249781,
                        -0.0910432, 0.0, 0.0, -1.1973854, 0.3995068,
                        0.0190365, 0.3995068, 0.0190365, 0.0, 0.0, 0.9634124,
                        0.0, 0.0, 0.0, -1.7856009, 0.0, 0.0, -0.1483901,
                        -0.8613751, 0.1483901, 0.8613751, 0.0641789,
                        -1.6968582, 0.0, 0.0, -0.1560517, 2.713923, 0.0, 0.0,
                        0.6847649, -0.3455308, -0.6459246, -0.3455308,
                        -0.6459246],
    "occupations": [2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0]
  },
  "properties": {
    "totalEnergy": -200549674.20668492
  }
}
```

The layout is designed to be compact, and follow the convention of using linear arrays where feasible to store sets of values. It also continues the index-based extension of atom properties with the `shellToAtomMap`. Chemical JSON is not meant to be used directly, but read into an appropriate data structure in the programming language of choice, this means that simple index-based look ups can be performed, but it may not be as human readable as desired. It will map to more efficient use of binary analogs such as BSON, jsonb, MessagePack and slightly less analogous HDF5 storage formats. The project has validated the approach and uses this format in C++, Python and JavaScript using in-memory representations, translation to other tools, and storage in MongoDB on the data server.

The data standards developed for the platform reuse community best practices, such as the work developing the IUPAC International Chemical Identifier (Heller et al., 2015), InChI key (a hashing standard built on InChI), and Simplified Molecular-Input Line-Entry System (SMILES) (*OpenSMILES Home Page*, n.d.). These standards provide simplified, unambiguous string representations of chemical structures (describing both elemental makeup and connectivity) that can be used to as common identifiers. Generating these and augmenting other data enables simple linking of data to existing databases such as PubChem (Kim et al., 2018), ChemSpider (Williams, 2011), Wikipedia, and more.

The Chemical JSON format forms the core of the data model in the platform described. It has an informal standardization process that would benefit from more discipline in the future. The development team worked with MolSSI from an early point in their founding to encourage the standardization of a flexible JSON-based format, and have actively participated in discussions. The Chemical JSON format is a useful internal representation, the documentation of which should be improved, but the ultimate goal is to offer seamless export to other formats such as those supported by current libraries in existing formats and new ones such as QCSchema. It is clear that for large data binary formats will be essential, and a number of them can be mapped to from JSON such as MessagePack and HDF5 as they offer a familiar container structure.

Flexible Data Server Platform

At the core of the project sits a RESTful data server, with a well-defined set of programming language-agnostic interfaces. These endpoints take the form of web URLs using the HTTP protocol underpinning the web. Simple queries are encoded in the address, with responses often returned as JSON. The approach is standardized and widely used, with software that can automatically generate documentation of the interfaces. The data server provides persistence of data, metadata, and workflow execution to clients such as the Jupyter notebooks (Kluyver et al., 2016; *Project Jupyter*, n.d.), desktop tools, command line clients, and other consumers of the platform.

The Girder project (*Girder: a data management platform — Girder 3.0.5 documentation*, n.d.) was used as the central component because it already offered a large amount of the more general functionality such as user authentication, file/directory storage, upload, download, and access control. It also offers a structured way to augment its functionality using plugins developed in Python. An early version of the data server was described in a previous publication along with the Python plugins developed (Hanwell et al., 2017). This has since evolved into a more ambitious project, but the data server concept described there is still at the core of the design. Existing capabilities in the framework such as authentication, access control, computational job control, and integration with HPC queuing systems offered clear advantages in this project.

The data server provides the central point of integration, it is data-centric and avoids direct generation of any HTML/visualization on the server. The RESTful APIs are consumed by different clients, principally the single-page web application, JupyterLab (*JupyterLab Documentation — JupyterLab 2.2.5 documentation*, n.d.) web widgets, and Python-based Jupyter kernels. The RESTful APIs are completely programming language agnostic, and can be consumed by others in the community either directly or by adapting reference examples developed as part of the project. We have developed a Python command-line client, Jupyter Python kernel extension, web widgets, and an Avogadro plugin to search and display data from the server.

The server platform forms the central component of the open software platform, providing persistence, authentication, metadata generation, coordination with microservices, and generation of new data. It has been developed to use a plugin to extend the server component with specific functionality, and critical services have been moved into specialized microservices such as an Open Babel microservice for file translation, 3D coordinate generation, chemical utility functions and an Avogadro microservice offering some file services and molecular orbital cube generation. They provide reference interfaces that can be improved or replaced as the need arises in future projects. Additionally the use of microservices enables the decoupling of services so that more compute intensive tasks can be scaled separately, for example 3D coordinate generation or cube generation.

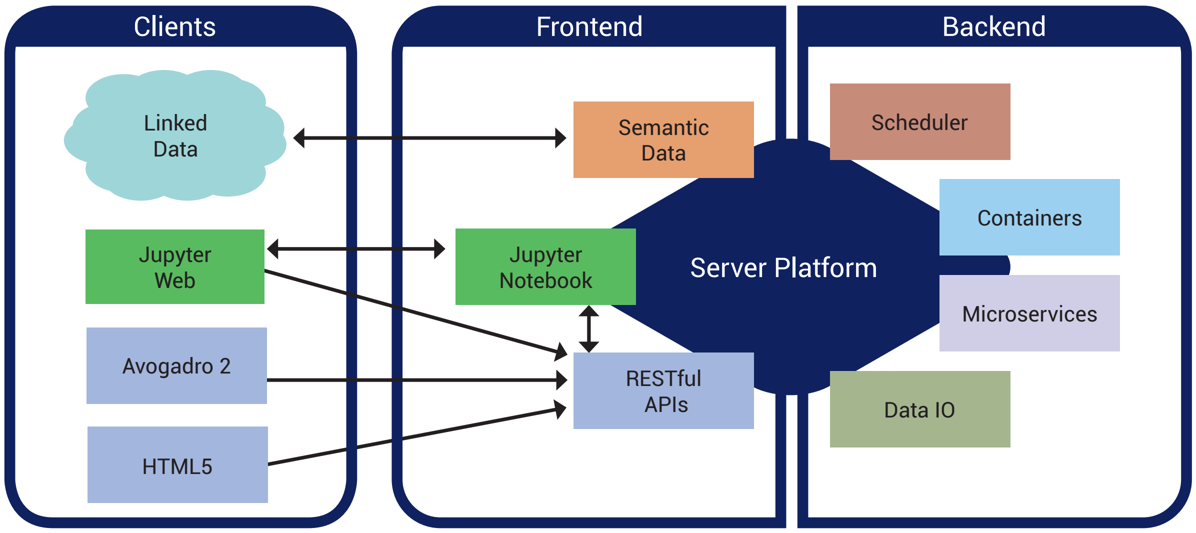


Figure 1: Overview of the platform's architecture, with the data server platform providing core services.

The high level software architecture was developed to enable researchers to pick and choose the most appropriate components, or to use the entire platform as a turnkey solution. The complete solution requires administrative privileges in order to run JupyterHub, but it can be run in a cutdown form where the Jupyter/JupyterHub integration is disabled, but prompts are offered so that users can run their own Jupyter notebook and connect to the data server. This works for most things, but removes the ability to store notebooks, and to link from calculations to the notebooks that generated them.

As the project has evolved it has become clear that whilst a complete solution provides a broader overview of the data generation, analysis, and publication process it can be too onerous. It may also be preferable to export as much of the process as possible to services that are able to provide greater guarantees of longevity, i.e. months, years, decades, centuries depending upon need. This led to adapting the project to serve in different roles where active data generation benefits greatly from a running data server. Whereas once projects are complete it may not be feasible to keep a project server up and running but there are a number of community services that can be employed to store static assets that clearly document the path from data generation to final figures such as Zenodo (*Zenodo - Research. Shared.*, n.d.), Binder (Jupyter et al., 2018) coupled with GitHub (*The Binder Project*, n.d.), and Figshare (*Figshare*, n.d.). To this end the data and metadata standards enable the export of individual calculations, and the notebooks can be archived. The

Binder export discussed later shows one example of moving to a static repository containing data and a notebook, the process is currently manual and we are working on improved automation.

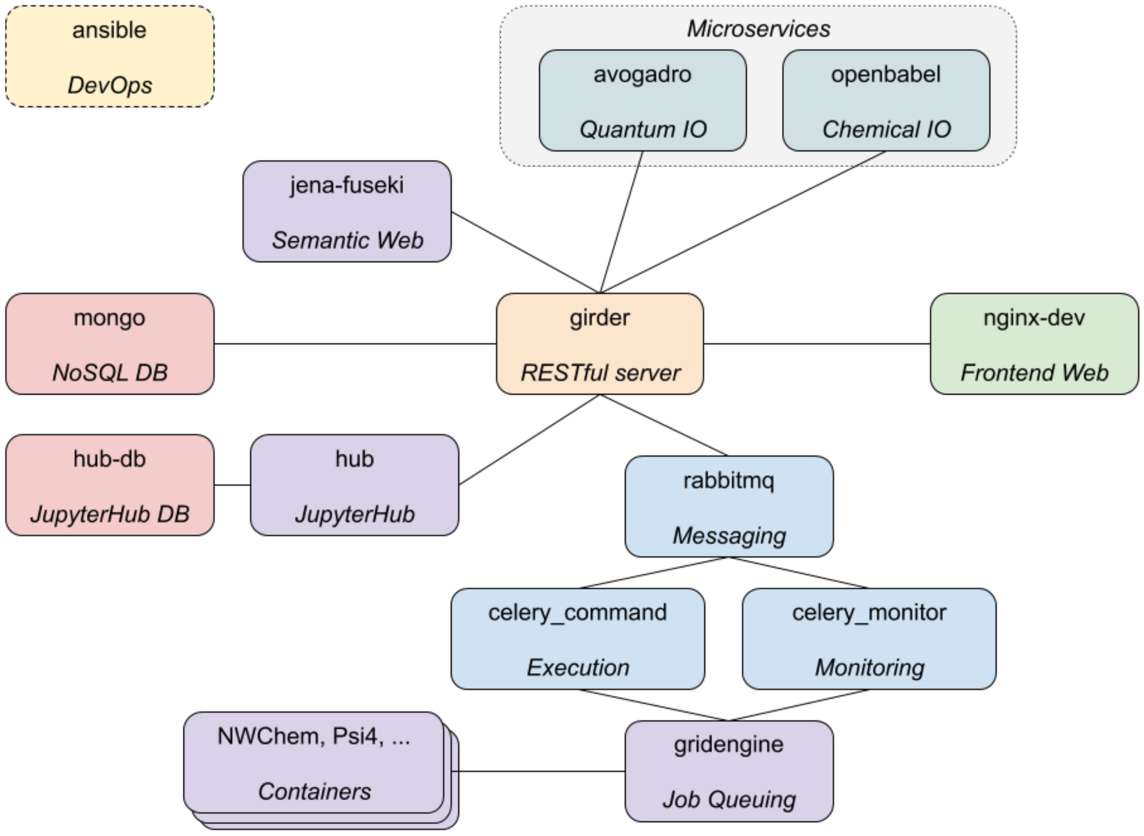


Figure 2: The containers for the complete platform, the "girder" container is the data server

The bulk of the code for the extensions developed for the Girder server are in the server repository (*OpenChemistry/mongochemserver*, n.d.), with deployment code in a separate deployment repository (*OpenChemistry/mongochemdeploy*, n.d.). They define the chemistry specific endpoints, and hook into the documentation capabilities provided by Girder and use MongoDB to persist data/metadata. Much more time could be spent on other aspects of the project by using a framework developed for scientific data, which also reuses many open source projects to provide the RESTful server, database access, schema validation, etc. Fig. 2 shows the containers that make up the platform, as deployed using Ansible, with the "girder" container hosting the data server, and providing integration of other components.

```
class Molecule(Resource):

    def __init__(self):
        super(Molecule, self).__init__()
        self.resourceName = 'molecules'
        self.route('GET', (), self.find)
        self.route('GET', ('inchikey', ':inchikey'), self.find_inchikey)
    @access.public
    def find(self, params):
        return MoleculeModel().find_molecule(params)

    @access.public
    def find_inchikey(self, inchikey, params):
        mol = MoleculeModel().find_inchikey(inchikey)
        if not mol:
            raise RestException('Molecule not found.', code=404)
        return self._clean(mol)
```

The code snippet above is taken from “girder/molecules/molecules/molecule.py” in the server repository, and shows an abbreviated version of the Molecule endpoint. The server can be extended with new endpoints such as this, where routes are added, and the capabilities of the underlying model can be exposed. Decorators define the access level required, for example “public” in this case for unauthenticated access, or “user” to indicate a user must be logged in. The model interacts with the database to find, update, delete, and amend records, providing an object-relational mapping (ORM) to map from the RESTful API to the underlying data representation.

```
class Molecule(AccessControlledModel):

    def __init__(self):
        super(Molecule, self).__init__()
        self.ensureIndex('properties.formula')
        self.ensureIndex('inchikey')

    def initialize(self):
        self.name = 'molecules'

    def find_molecule(self, search = None):
        limit, offset, sort = parse_pagination_params(search)

        if search is None:
            search = {}

        query = {}
        if 'queryString' in search:
            # queryString takes precedence over all other search params
            query_string = search['queryString']
            try:
                query = mol_query.to_mongo_query(query_string)
            except mol_query.InvalidQuery:
                raise RestException('Invalid query', 400)
        elif search:
            # If the search dict is not empty, perform a search
            if 'name' in search:
                query['name'] = { '$regex': '^' + search['name'],
                                  '$options': 'i' }
            if 'inchi' in search:
                query['inchi'] = search['inchi']
            if 'inchikey' in search:
                query['inchikey'] = search['inchikey']
            if 'smiles' in search:
                # Make sure it is canonical before searching
                query['smiles'] = openbabel.to_smiles(search['smiles'], 'smi')
            if 'formula' in search:
                formula_regx = re.compile('^%s$' % search['formula'],
                                          re.IGNORECASE)
                query['properties.formula'] = formula_regx
            if 'creatorId' in search:
                query['creatorId'] = ObjectId(search['creatorId'])

        fields = [
          'inchikey',
          'smiles',
          'properties',
          'name'
        ]

        cursor = self.find(query, fields=fields, limit=limit, offset=offset,
                           sort=sort)

        num_matches = cursor.collection.count_documents(query)

        mols = [x for x in cursor]
        return search_results_dict(mols, num_matches, limit, offset, sort)

    def find_inchikey(self, inchikey):
        query = { 'inchikey': inchikey }
        mol = self.findOne(query)
        return mol
```

A truncated example of the corresponding model is shown above. The `find_molecule` function is quite complex, supporting multiple fields, pagination, etc. These are necessary to support larger result sets where returning all results would take too much space, and would not be desirable. It enables clients to display pages, and request offsets into the results returned by a search query. The `find_inchikey` is one of the simplest functions where a query is created, and the query is executed with the result returned directly. There is a lot of freedom within the model to add chemistry specific logic, or perform very simple database queries depending upon the need. This also introduces a separation of concerns where the model code interacts with the database, whereas the endpoint code deals with issuing the correct response, documenting endpoints, and dispatching to the appropriate parts of the data model.

Having an open source data server not only offers useful capabilities now, but through the use of Python, and by documenting the code using modern approaches, it offers a starting point for future projects that want to extend an already capable data server. The use of microservices works toward making this even easier by providing containers that perform specific tasks that could be reused within other platforms, or where their capabilities can be extended and improved within this platform without the need to work within the data server itself. Both microservices were migrated from the server component to simplify the data server, and offer greater modularity. The Open Babel and Avogadro microservices were able to separate out longer running tasks from the data server thread ensuring that the data server is less likely to stall and the microservices can be scaled more easily.

Containers for Chemistry Codes

The use of software containers has revolutionized how services are deployed on the internet, and science is beginning to reap some of these rewards. At their core, containers such as Docker(Merkel, 2014), Singularity(Kurtzer et al., 2017), and Shifter(Gerhardt et al., 2017), are self-contained Linux-based binary environments that contain everything from Linux system calls through to the full call stack of software libraries. This enables the development of a complete build environment from a known base image, through to packages that should be installed, software repositories to clone, build, and install.

The software images have an entry point that can be called from the host operating system, and a flexible set of capabilities to mount file systems, get data in and out, etc. The use of a Dockerfile enables a full description of the base image, build steps, and configuration. The binary images can be uploaded to DockerHub and similar services, and downloaded using standard tools developed by the community. The container specification developed for the project offers a standard method of invoking a code, with input directives delivered in a simple JSON format, geometry using Chemical JSON input, and output translated to a simple JSON output format or Chemical JSON. All relevant files are in the "docker" folder in the deployment repository (*OpenChemistry/mongochemdeploy*, n.d.), with a folder for each of the containers.

Docker was the first widely adopted container standard, and it acts as a base for the images used in this project. It provides network isolation, which is not necessary for most computational chemistry tasks or HPC tasks in general. The Singularity project was founded to address this issue, concentrating more on the execution of binary programs without the need for privileged system access often not available on HPC clusters/supercomputers. Shifter offers a solution developed at NERSC for executing containers within their supercomputing environment. If Singularity were available on all environments that would probably be the only container format used, but at the present time this is not the case and so multiple container types are supported depending upon deployment requirements. The Docker images are used to generate the Singularity and Shifter containers, so that it is only necessary to maintain a single image for each code.

The containers developed to support the project go beyond providing a useful delivery mechanism for executable code. They reduce the barriers to collaboration, and enable binary containers to be used as reference implementations that can easily move between host systems. Private container registries can host images containing proprietary software, and automated builds can enable sharing codes that have complex dependencies with build systems that can hamper seamless testing of changes. As supercomputers, HPC and cloud environments evolve the utility of containers will only make these advantages clearer to the wider community.

Quantum Chemistry Codes

Thanks to the container infrastructure described above, it is easy to extend the platform to perform calculations using new computational chemistry packages. Assuming the general flow follows that of an existing code the Dockerfile and Python driver can be used as a starting basis. If the code is packaged using a Linux distribution, or a Python packaging repository they can simply be installed as part of the image build process. The Python driver script provides a basis for adapting input data and directives to input files for the code, and translating output data to a suitable form that can be consumed by the platform. Once these pieces have been implemented the container can be tested on the command line before attempting to use it on the server-side.

The NWChem(Valiev et al., 2010) and Psi4(Parrish et al., 2017) packages were used as our first computational chemistry codes due to their popularity, open source licensing, and the fact that they offer similar features at a high level with very different implementations. These images serve as an example for others interested in adding new codes to the platform. The NWChem and Psi4 images support a similar set of features:

- Single point energy, geometry optimization and frequency calculations
- Basis set selection
- Theory level (Hartree Fock, DFT, MP2) and exchange-correlation functional selection

- Charge and multiplicity

This only represents a select subset of what these quantum chemistry codes are capable of. The features exposed by the images will be extended in the future, prioritizing things that are relevant to the needs of our platform's users. The input mechanism through the notebooks and API accepts arguments that are passed through to the driver script making it easy to add custom arguments to new containers. Container names consist of an organization, name and version currently pulled from DockerHub, making it easy to add custom versions and fetch them.

```
FROM conda/miniconda3:latest

RUN apt-get update && \
  apt-get install -y gcc git && \
  rm -rf /var/lib/apt/lists/*

RUN conda update --all && \
  conda install psi4=1.2 psi4-rt=1.2 python=3.6 -c psi4

COPY requirements.txt /app/
RUN pip install -r /app/requirements.txt

ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8

COPY src/* /app/

ENTRYPOINT ["python", "/app/main.py"]
```

The Psi4 container is one of the simplest, it uses the Python 3 Conda based image as a base, and then installs Psi4 from the Conda package provided by the Psi4 project. The Python driver script is then set as the entry point, and acts as the gatekeeper for getting data into and out of the container. When new releases are made updated images can be made. The NWChem container is a little more complex as it clones the git repository of a fork of the NWChem codebase that supports JSON output, it then uses Avogadro to translate that output to Chemical JSON. An example of running one of the containers is shown below.

```
docker run -v $(pwd):/data openchemistry/psi4:latest -g /data/geometry.xyz -p /data/parameters.json -o /
```

The JSON used as input follows a simple format that is translated by the driver script into the appropriate input parameters. One of the simplest examples is shown below, which can be used with an input geometry to drive the container on the command-line for testing purposes (or even use within other platforms).

```
{
  "theory": "hf",
  "task": "energy",
  "basis": "6-31g",
  "charge": 0,
  "multiplicity": 1
}
```

These input parameters have been added to the output Chemical JSON in order to make it easier to see what calculation parameters were used. The keys are common across the codes already present, but it is up to the driver script to decide how to interpret them. The "psi4" in the image name can be substituted with "nwchem" keeping all other arguments/JSON input the same and an NWChem calculation will be executed. Existing containers aim for consistency, but casting an eye to the future the platform will support the development of new methods, or the extension of existing containers beyond their current capabilities.

Machine Learning

From the point of view of the platform, there is no difference between running a quantum chemistry or a machine learning code. They both take as input a molecule and some parameters, and return a conforming output. The main difference is that machine learning codes may not require the existence of a 3D structure, and can still operate when only string based formats such as InChI or SMILES are available. The server was extended to support the addition of molecules that have no 3D structure so that machine learning can be used more effectively within the platform as 3D coordinate generation can be quite expensive when considering high-throughput models.

If many small calculations (especially machine learning calculations on line format structures) are to be performed on a large set of molecules, it can be very inefficient to have a task monitor and a docker container for each individual calculation. For this reason, the user is able to submit a batch of calculations as a single task. In this task, there is only one task monitor and one docker container used, and the docker container is given the freedom to choose how to run the calculations (one at a time, one per core, etc.). Additionally, each calculation is checked to see if it has been performed before. If so, it will skip that calculation and use the output already present in the database.

Images were developed for ChemML(Haghighatlari et al., 2020; Hachmann et al., 2018) and ANI(Smith et al., 2017). These represent two very different classes of machine learning models. The illustrated model developed by ChemML package takes in a SMILES string, and predicts corresponding properties which are output as key-value pairs. ANI behaves more like a typical quantum chemistry code that takes in 3D chemical structure, can perform an energy calculation, geometry optimization, or frequency calculation and output (potentially updated) 3D chemical structure. It doesn't require as much input as is typical for other codes, and it doesn't output electronic structure. Both required further generalization of our treatment of tasks in order to adapt to their unique requirements, and added examples of quite distinct types of model paving the way to easily add more using similar patterns.

ChemML is a general-purpose machine learning package developed in Python for the analysis and modeling of chemical and materials data. Although ChemML provides essential tools to develop and train machine learning models in general, we aim to deploy trained models and broaden their utilization in the chemistry community. Our long-term goal is to compile a collection of predictive models for certain properties that can be used as alternatives to corresponding physics-based modeling or simulation approaches. To start, we developed a deep learning model for prediction of the refractive index values of organic compounds and made it available as a Docker container.

The model takes as input the SMILES string of an organic molecule, and returns as output a few quantities of interest that the model has been trained on. The predictions are available almost instantaneously (unlike a physics-based model via the Lorentz-Lorenz equation parametrized by inputs from quantum chemistry and molecular dynamics calculations). The results of the model are comparable with those of other data-derived prediction models in terms of diversity of molecular candidates and the accuracy of predictions. The current implementation enables the user to retrain each model for a better or more generalized prediction power. Moreover, a trained model can leverage other relevant ML models through the concept of transfer learning design methodologies. Our near-term plans for future development include the addition of models trained on other data sets and for other material properties.

The ANI image used the PyTorch(*PyTorch: An Imperative Style, High-Performance Deep Learning Library*, n.d.) implementation of the ANI potentials - TorchANI (Gao et al., 2020). We make use of the algorithms in ASE to drive tasks such as geometry optimization and normal mode calculations. The TorchANI image features the *ani-1x* and *ani-1ccx* optimized potentials that can be used to generate single point energies, perform geometry optimizations and compute Hessians for frequency calculations. This makes the ANI image look quite similar to the quantum chemistry codes, but no basis set selection is necessary and there is only support for a limited set of chemical elements.

For the elements that are supported ANI offers superior performance to geometry optimizations performed using conventional quantum chemistry codes, often trained on a higher level of theory than would typically be used. It also offers performance comparable with molecular mechanics codes without the need for atom typing and other parameterization. This often makes a geometry optimization desirable before using other codes assuming the chemical structure only uses the elemental set supported. It is also possible to use other techniques and compare the output in order to validate its use for particular chemical systems.

Generating and Accessing Data on the Web

One of the primary goals of this project is to make computational chemistry tools more accessible, and to enhance their use within a collaborative online environment where data can be generated within the web browser. This does not preclude the use of existing tools, but it provides an alternative way of accessing tools that have seen significant investment using one of the most pervasive mediums in the modern world. The use of web-based tools makes the platform accessible on low powered phones through to the most powerful desktop workstations. The platform itself can provide access to calculations already run almost instantaneously, while enabling authenticated users to generate new data.

The previous section describes the non-graphical server-side capabilities developed for the software platform. These are necessary tools, but they do not offer an approachable user interface for generating, interacting with, analyzing and visualizing data. This section discusses a number of capabilities developed to offer web-based interfaces to look at data, search it, categorize it, and in many cases these components can be used independently of the complete platform. The high-level web technologies used include JavaScript and TypeScript for code development, WebGL for 3D visualization, and React as a framework for exposing elements in web applications.

Web Widgets

A set of reusable widgets have been developed that can be embedded in any web environment, from a React (*React – A JavaScript library for building user interfaces*, n.d.), Angular (*AngularJS — Superheroic JavaScript MVW Framework*, n.d.) or Vue (*Vue.js*, n.d.) based single page web application, a JupyterLab extension, or a static HTML page. They were developed in the TypeScript language using the Stencil library (*Stencil*, n.d.). Once they have been transpiled into JavaScript they become standard web components that can be used just like any other HTML tag such as a `<div/>` or a `<img/>`.

The core widgets are exposed using the `<oc-molecule-moljs/>` and `<oc-molecule-vtkjs/>` tags. The widgets have a common interface and can be used interchangeably. They take Chemical JSON as input, and a set of parameters can be used to tweak the visualization (such as the isosurface value, the active normal mode, or the molecule representation type). They render a three dimensional representation of the molecule that the user can interact with using the mouse or touch-based interaction, such as the 3Dmol.js based widget shown in Fig. 3. The difference between them is that `<oc-molecule-moljs/>` uses 3Dmol.js (Rego & Koes, 2015) to render the 3D scene, while `<oc-molecule-vtkjs/>` uses vtk.js (*vtk.js*, n.d.).

The major features of the widgets are the display of the molecular structure and connectivity, animation of normal modes, and the display of isosurface data from a cube of data (typically molecular orbital data from a quantum chemical calculation). The vtk.js based widget can display the crystal unit cell and offers volume rendering of cube data, shown in Fig. 4, in addition to isosurface rendering.

The `<oc-molecule/>` tag wraps the core `<oc-molecule-moljs/>` and `<oc-molecule-vtkjs/>` and adds a menu so that the user can easily change the visualization parameters. The menu (click on the three dots in the right top corner of the figure) offers:

- Structure representation (ball and stick, Van der Waals. or custom)
- Select the rendering library to use (3Dmol.js or vtk.js)
- Selection of the active normal mode for vibrational mode visualization

- Selection of the isosurface value for molecular orbital visualization
- Selection of the color map used by the volume renderer
- Modification of the opacity function used by the volume renderer

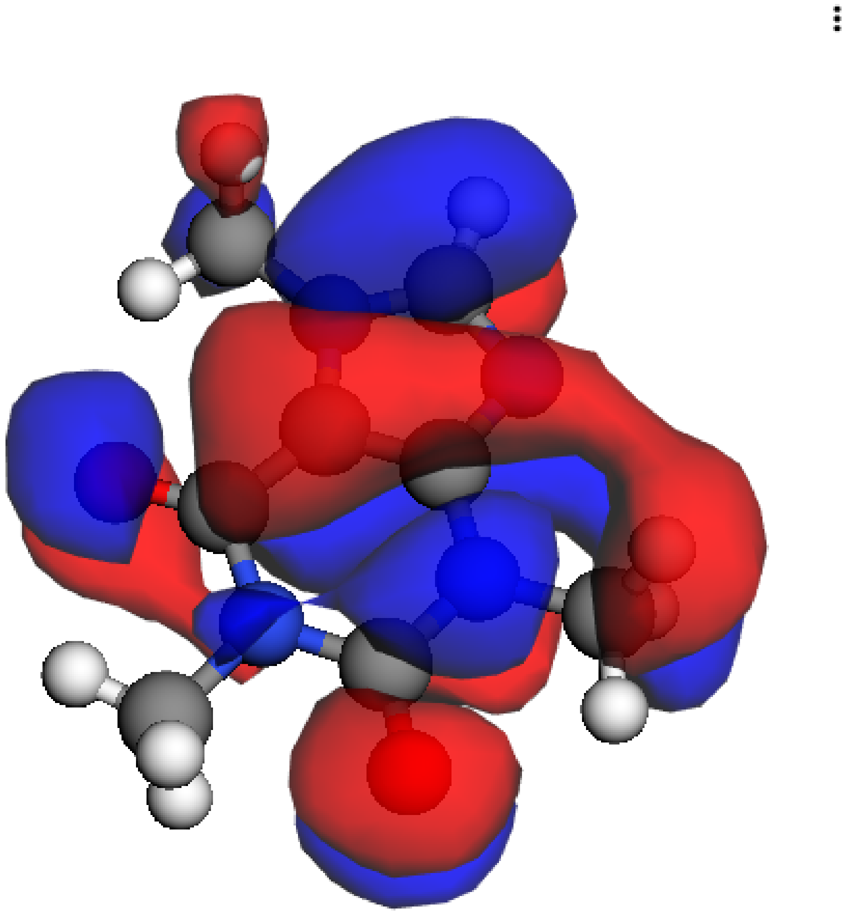


Figure 3: An interactively rendered caffeine molecule saved in a Chemical JSON file with a molecular orbital rendered using the 3Dmol.js based backend.

Jupyter Extensions

JupyterLab is the new web frontend for the Jupyter project. It can be extended using a mime renderer extension, this has been used to serve two purposes: displaying interactive visualizations of molecules in the output of a notebook cell, and monitoring computational jobs being executed through the platform. The decision was taken at an early point in the project to support JupyterLab extensions exclusively, although it would certainly be possible to add support for the older Jupyter web frontend if there was a big call for it.

The visualization of the molecular structure (see Fig. 3 and 4) is accomplished by using the `<oc-molecule/>` component described earlier. The widget is displayed when the user opens a `cjson` file in JupyterLab, or when selecting the result of a calculation as the output of a notebook cell. Thanks to some utility functions in

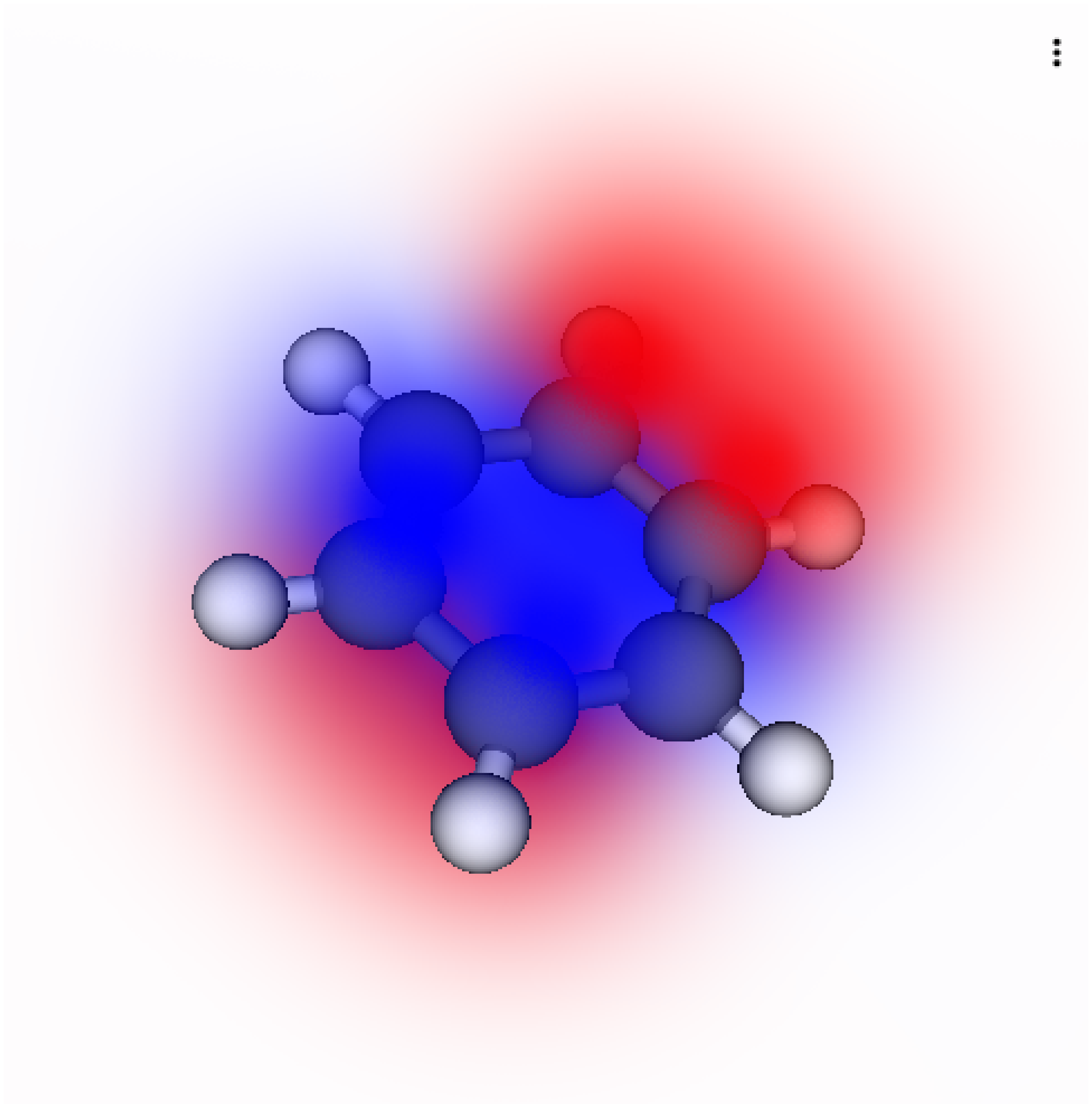


Figure 4: An interactively rendered benzene molecule saved in a Chemical JSON file with a molecular orbital rendered using the vtk.js based backend.

the `openchemistry` Python package, the molecular visualization can be used for arbitrary Avogadro molecule objects. So, no matter the provenance of the molecular structure, if the user is able to convert it into an Avogadro molecule they will be able to take advantage of this JupyterLab extension to get interactive 3D visualizations in the notebook.

A JupyterLab notebook is the preferred environment to dispatch and analyze calculations performed through the platform. For this reason part of the extension offers a widget to monitor running jobs. The monitor widget (see Fig. **??**) displays a table with a row per job, and information such as the Docker image running the job, the version, and the current status. If a job is selected, it is possible to see log messages and error messages in real-time. The job monitoring widget will also instruct the user to execute the cell again to see the result once the job has completed, or it will display a failure condition if things did not go as planned.

In addition to the JupyterLab web extensions described an openchemistry Python package was developed to extend the Jupyter kernel running on the server. It uses the same RESTful API endpoints that were described in the previous section to query the data server, initiate new calculations, and monitor progress. The module also has the utility functions described so that it can work independently using local data, or take environment variables to run the notebook kernel locally and communicate with a remote data server. This results in a flexible deployment strategy where a fully integrated server can invoke a notebook with all the right environment variables set up, a local (or centralized) notebook server can be invoked with the right environment variables set, or a static notebook/local notebook can avoid using the data server and refer to local data files.

Rich media available at `https://youtu.be/7otfejlAoqg`

The execution of notebooks and the execution of containers are two of the most demanding aspects of the project in terms of infrastructure and computational resources required. They can both be disabled, resulting in a data server that offers data query, retrieval, and visualization capabilities. They both allow a great deal of freedom to execute code on a shared resource, and so must be considered carefully in terms of offering access due to the security considerations and resource constraints.

There is a balance to strike between what is executed in the notebook's kernel, what is executed by the data server (including its microservices), and what is sent to the queuing system for batch execution. Some things may be duplicated, but we have tried to create a balance based upon the task. When things are executed in the Python kernel they tend to be relatively short-lived tasks, or calls to the data server. The data server is able to cache results so that they can be retrieved in other notebooks, over the web interface, or using other methods without executing each time, such as 3D coordinate generation using Open Babel, or molecular orbital calculation. Longer lived geometry optimizations, executing quantum codes, and running machine learning kernels are usually performed using the queuing system so that they can use HPC resources and images.

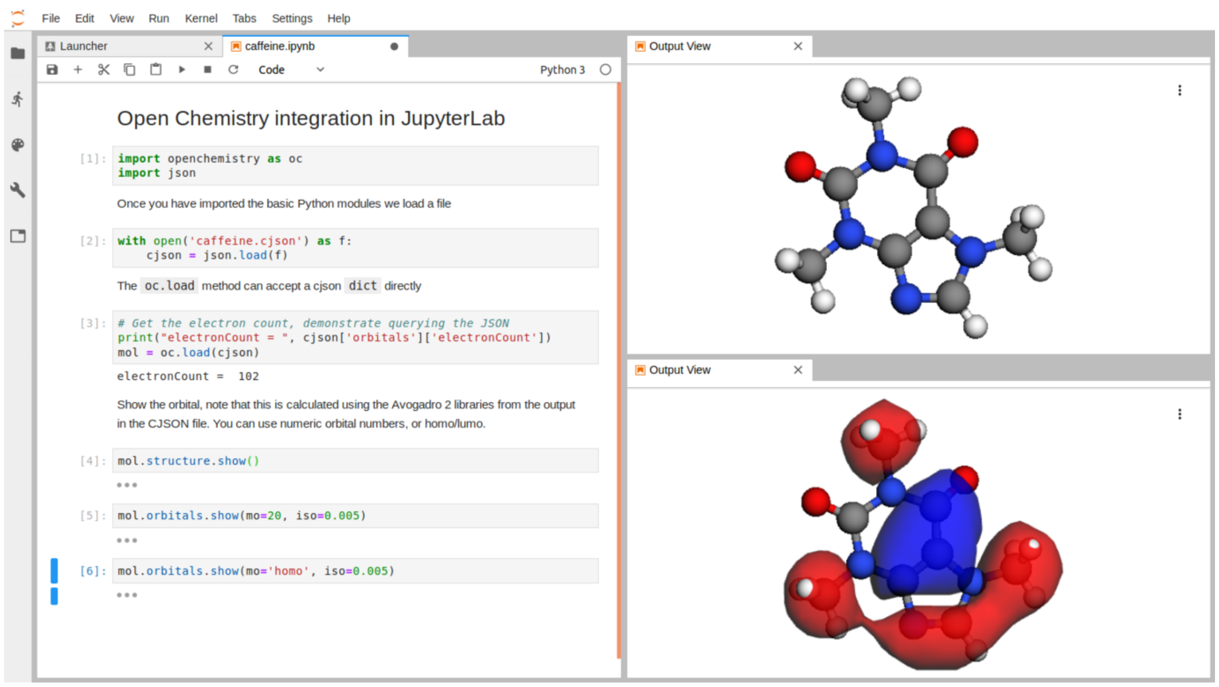
Export to Binder

The Binder project has led to the development of reproducible Jupyter notebooks that can be executed, changed, and examined for free on the web. A GitHub repository can be used to specify the environment used, including what Python modules are needed for the Jupyter kernel and what extensions should be installed for JupyterLab before the notebook opens in the browser. A Docker container is generated on the fly in response to the specification in the repository, and once ready the repository is mounted inside the container and the notebook opened.

This is clearly a powerful tool that is ready for use by the wider community, and offers a path to providing full access to the data and notebooks for anyone without the need to create accounts. The Python module that had been developed to extend the Jupyter kernel was extended to support execution without a data server, and packaged using pip so that it could be installed via the mechanism offered by the Binder project. A blog post was written in December, 2018 describing the approach (*Avogadro and Quantum Chemistry Viz in JupyterLab - Kitware Blog*, n.d.), and showing the Binder repository with example links to data for calculations (*OpenChemistry/jupyter-examples*, n.d.).

HTML5 Web Application

The main entry point to to the platform is a state-of-the-art web application developed. It offers a landing page, with anonymous access to public data, authentication, and access to notebooks, molecules, calculations



Open Chemistry integration in JupyterLab

```
[1]: import openchemistry as oc
     import json
```

Once you have imported the basic Python modules we load a file

```
[2]: with open('caffeine.cjson') as f:
         cjson = json.load(f)
```

The `oc.load` method can accept a cjson `dict` directly

```
[3]: # Get the electron count, demonstrate querying the JSON
     print("electronCount = ", cjson['orbitals']['electronCount'])
     mol = oc.load(cjson)

     electronCount =  102
```

Show the orbital, note that this is calculated using the Avogadro 2 libraries from the output in the CJSON file. You can use numeric orbital numbers, or homo/lumo.

```
[4]: mol.structure.show()
     •••
[5]: mol.orbitals.show(mo=20, iso=0.005)
     •••
[6]: mol.orbitals.show(mo='homo', iso=0.005)
     •••
```

Figure 5: The JupyterLab extension allows the visualization of molecular structures in a notebook, with Binder used here to offer a static repository to offer a more interactive exploration of data.

and profile services once authenticated. Both the AWS instance using Google's OAuth provider and the NERSC instance integrate with their authentication provider negating the need to create new accounts. The profile page can be used to associate Twitter and/or ORCID identifiers with an account, and access to a summary of the notebooks, molecules, and calculations authored/initiated by the account's owner.

Basic search is offered to filter structures and calculations based on the molecule name, InChI, or SMILES. Internal navigation is possible, cross linking of all calculations associated with a specific molecule and vice versa. In the detailed page of a structure or calculation there are links to download the data in various formats, such as Chemical JSON, XYZ, SDF, or the raw output of the simulation code.

The application is written using the React + Redux framework in a mixture of JavaScript and TypeScript. The server component is completely decoupled from the application, and communication between the two only happens via the REST API. For the 3D visualization elements we rely on the reusable `<oc-molecule/>` component that was discussed earlier, with the same look and feel as in the Jupyter notebooks and the static web page examples. They are coupled with the search, linking, and other facilities to provide an interactive view of the data present on the instance being explored.

Rich media available at `https://youtu.be/5nUff6O_Gf0`

Fig. **??** shows a demonstration of the main web application demonstrating how anonymous users can access public data, and authenticated users can do more once they supply their credentials. In both cases it is possible to see the notebook that generated the data, but for anonymous users that is a read-only view whereas authenticated users will be able to launch the notebook. The molecular data, and calculation results can be downloaded from the web application using buttons on the relevant pages, with permanent URLs identifying specific molecules, calculations, and even the orbital/contour being visualized. These views of the data are designed to be simpler, more focused and use responsive elements so that they can be viewed on mobile telephones up to full desktop systems.

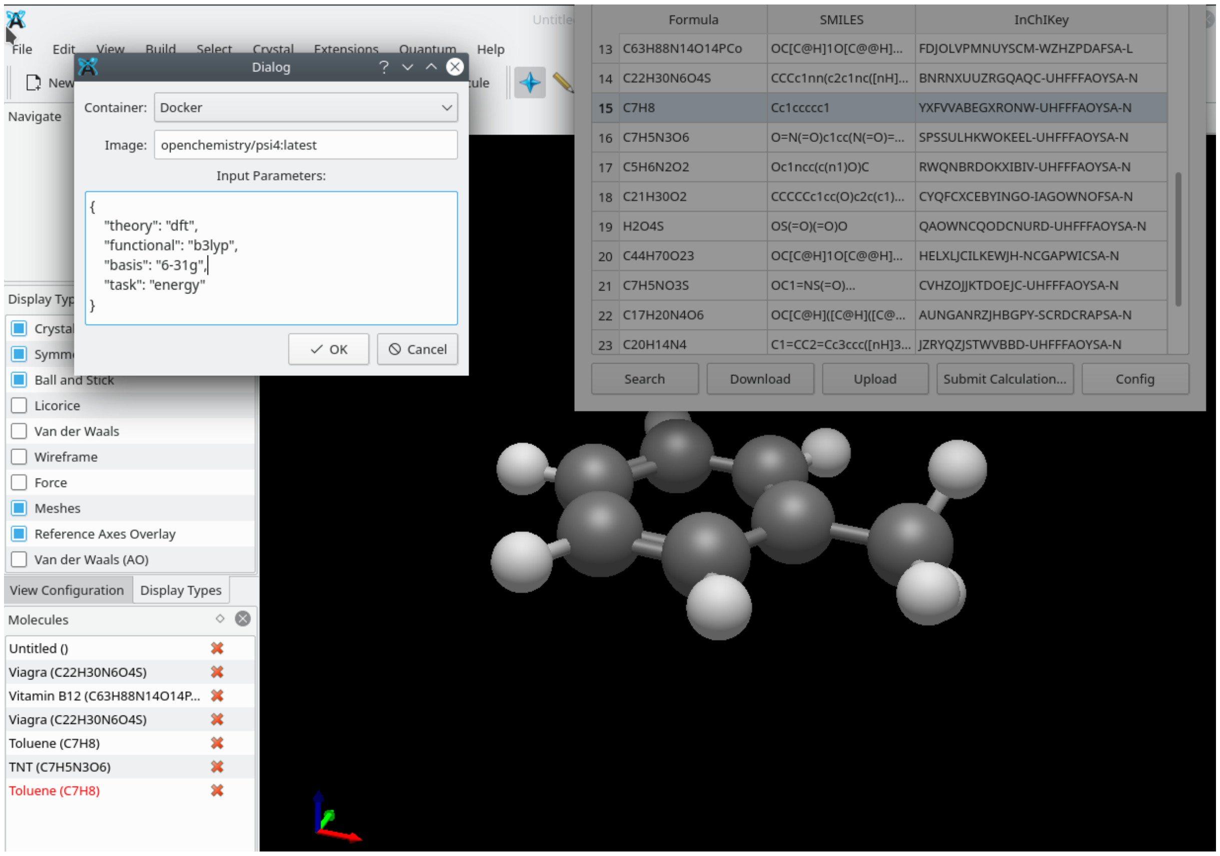
Extensions to Python and Avogadro

The server platform has an extensive RESTful API that is used by the web widgets, Jupyter components, and web application. These same interfaces have been reused in a Python client that can interact with the server using an API key, offering simple job monitoring, structure upload, etc. This gives a simple path to upload a number of existing files from the command line in an automated fashion, or to augment existing workflows with upload to a data server in order to benefit from the capabilities provided by this platform.

The Avogadro application has made use of web databases for many years. These have been read-only, and offered an intuitive way to search and import existing structures. This includes searching the Protein Data Bank, or the Cactus structure resolution services (*NCI/CADD Chemical Identifier Resolver*, n.d.) where users can simply type the name of a chemical to search for and retrieve a structure if found. These capabilities were very popular with users, but offered no ability to share results, or generate new data that was made available through a chemical data platform.

A plugin has been developed for the Avogadro application that can perform basic searches of the data available. Far more interesting is the ability to use an API key for an authenticated user to upload chemical structures, and request calculations. This makes a host of capabilities available to the desktop tool, and readily enhances all data passing through the platform by adding additional metadata, search, and discoverability. The Avogadro tool can be used to download a structure, or drawing one freehand, with basic geometry optimization available locally using Open Babel. The user can then upload the structure to the server, making it available on the web frontend, and they can request a calculation using the plugin interface (Fig. 6). The result can be downloaded and visualized when ready, or the user can share a link to the record created and share it more widely on their local intranet, or the wider internet if using a publicly hosted instance.

The plugin can switch between instances of the data server, with future enhancements planned to offer a selection of one or more servers, and basic interaction capabilities. This could then search across accessible instances to find existing data, and accelerate workflows by making data more easily available. The addition of unique identifiers enables users to retrieve data based on an unambiguous identifier, so that coordinates can be edited efficiently, conformations examined, and more before uploading a performing calculation on the selected geometry. These can then be linked to directly using identifiers, or searched across sites using InChI, InChI key, SMILES, etc.



Container: Docker

Image: openchemistry/psi4:latest

Input Parameters:

```
{
   "theory": "dft",
   "functional": "b3lyp",
   "basis": "6-31g",
   "task": "energy"
}
```

	Formula	SMILES	InChIKey
13	C63H88N14O14PCo	OC[C@H]1O[C@@H]...	FDJOLVPMNUYSCM-WZHZPDAFSA-L
14	C22H30N6O4S	CCCc1nn(c2c1nc([nH]...	BNRNXUUZRGQAQC-UHFFFAOYSA-N
15	C7H8	Cc1ccccc1	YXFVVABEGXRONW-UHFFFAOYSA-N
16	C7H5N3O6	O=N(=O)c1cc(N(=O)=...	SPSSULHKWOKEEL-UHFFFAOYSA-N
17	C5H6N2O2	Oc1ncc(c(n1)O)C	RWQNBRDOKXIBIV-UHFFFAOYSA-N
18	C21H30O2	CCCCCc1cc(O)c2c(c1)...	CYQFCXCEBYINGO-IAGOWNOFSA-N
19	H2O4S	OS(=O)(=O)O	QAOWNCQODCNURD-UHFFFAOYSA-N
20	C44H70O23	OC[C@H]1O[C@@H]...	HELXLJCILKEWJH-NCGAPWICSA-N
21	C7H5NO3S	OC1=NS(=O)...	CVHZOJJKTDOEJC-UHFFFAOYSA-N
22	C17H20N4O6	OC[C@H]([C@H]([C@...	AUNGANRZJHBGPY-SCRDCRAPSA-N
23	C20H14N4	C1=CC2=Cc3ccc([nH]3...	JZRYQZJSTWVBBD-UHFFFAOYSA-N

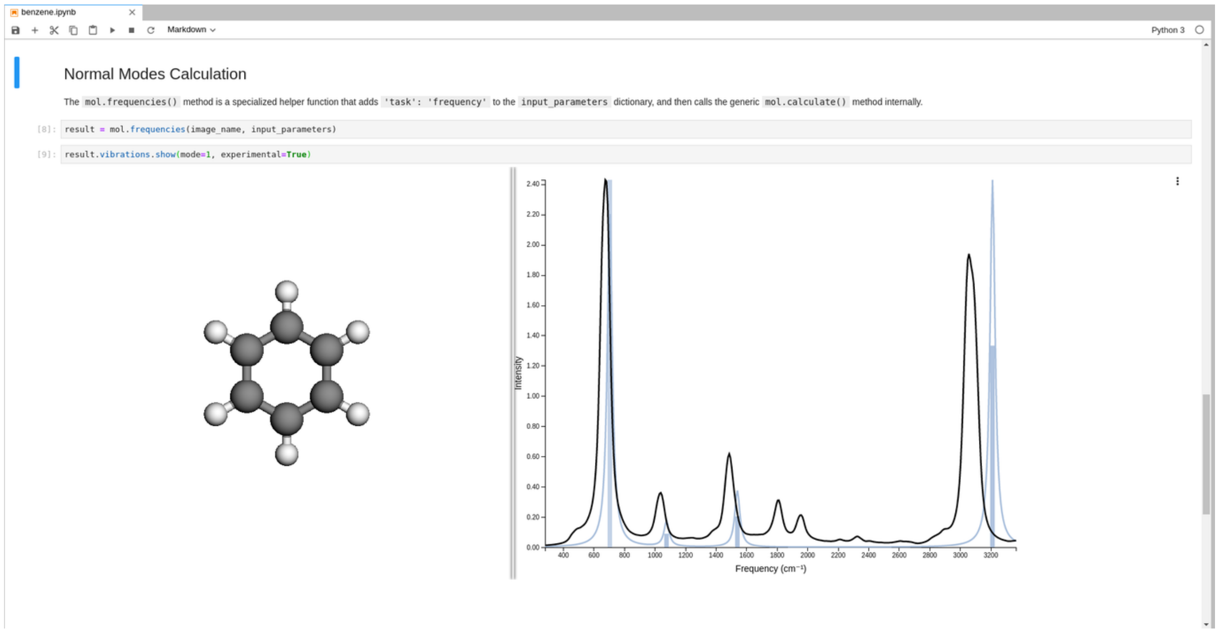
Molecules
Untitled ()
Viagra (C22H30N6O4S)
Vitamin B12 (C63H88N14O14P...
Viagra (C22H30N6O4S)
Toluene (C7H8)
TNT (C7H5N3O6)
Toluene (C7H8)

Figure 6: The Avogadro plugin showing basic search, along with a simple calculation dialog able to submit jobs to the server.

Integration of experimental data

Over the last decade the electronic structure community has refined their methods to make them more computationally efficient and accurate. As a result from computational chemistry calculations are starting to be on par with experimental results(Solà et al., 2019; Sperger et al., 2016). This revolution has opened the path for experimentalists and theoreticians to integrate their data sources and establish more consistent experiments and theories. Providing a platform to access scientific data from both worlds is therefore important and useful. Computational chemistry calculations can be used to examine a hypothesis before carrying out the experiments saving time and money, and they can serve to interpret the experimental observations. On the other side, access to experimental results enables the improvement of theories and methods(Mata & Suhm, 2017).

The Open Chemistry platform allows the calculation of theoretical spectra using the provided quantum chemistry containers. For instance, the prediction of theoretical infrared spectra (IR) can be routinely computed by quantum chemistry packages. Users can request the calculation of a theoretical IR with a chosen level of theory, and visualize it within the notebook. Additionally, a keyword argument can be passed to overlap experimental data queried from the NIST Chemistry WebBook database(*NIST Chemistry WebBook*, n.d.). In Fig. 7, we show a demonstrative example of an overlap between theoretical and experimental infrared spectra within a Jupyter Notebook in Open Chemistry. When visually inspecting the spectra to carry out the vibrational band assignment, we identify an intense IR active peak at about 750 cm^{-1} that corresponds to the characteristic absorption of C-H bending in aromatic compounds. A broad active IR peak is also present in the range of 2800-3200 cm^{-1} that corresponds to C-H stretching. Finally, the typical signals in the range 1500-1700 cm^{-1} are characteristics absorption of C=C bending. Having the ability to make band assignments together with experimental IR spectrum and theoretical animated vibrational modes is a feature that eases this task.



Normal Modes Calculation

The `mol.frequencies()` method is a specialized helper function that adds `'task': 'frequency'` to the `input_parameters` dictionary, and then calls the generic `mol.calculate()` method internally.

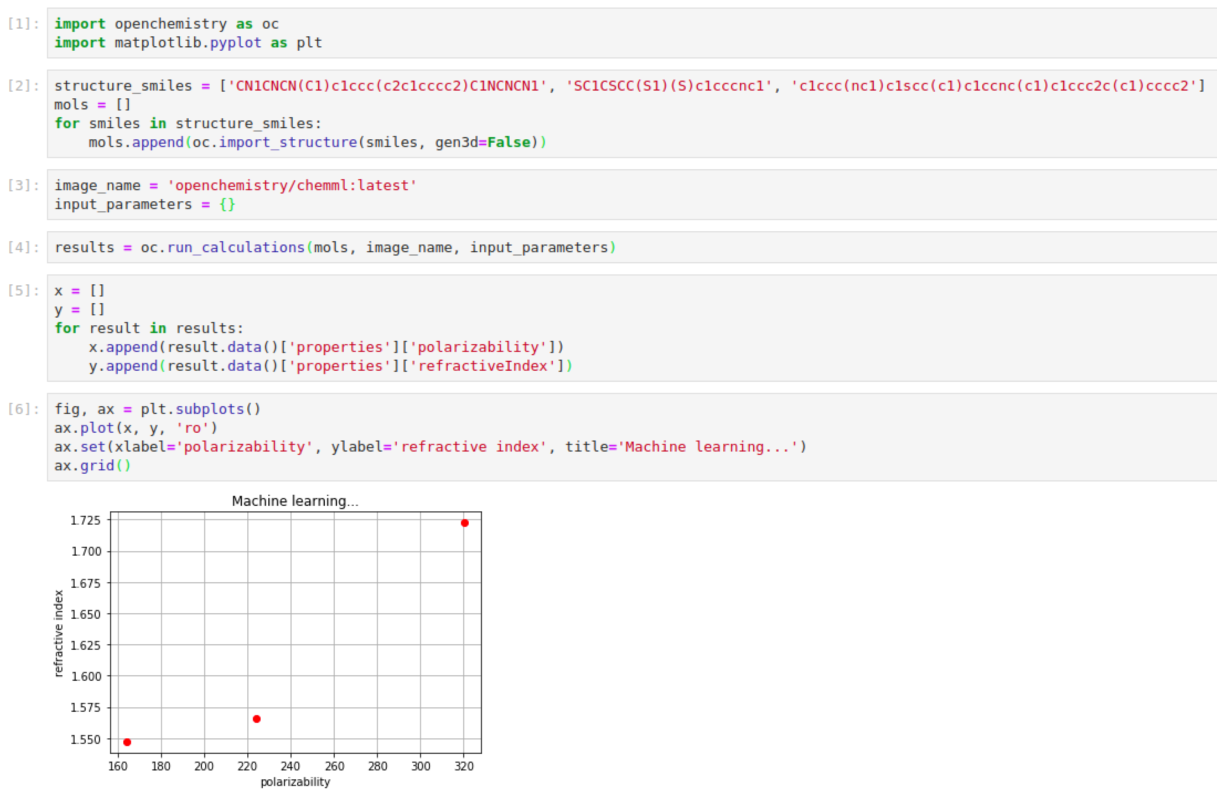
```
[8]: result = mol.frequencies(image_name, input_parameters)
```

```
[9]: result.vibrations.show(mode=1, experimental=True)
```

Figure 7: Overlap of experimental infrared spectrum of benzene over theoretical results.

Data analytics and mining

Coupling a data server to a computational notebook like Jupyter offers the possibility of performing analytics on the data generated and compiled within the platform. The platform allows us to import classes of compound, run various tools in the batch queuing system, and parse out important information from the outputs. Once a set of results is available, the notebook environment of the platform and its binding to ChemML allows users to analyze and mine the data in its entirety, rather than only referring to individual results. This statistical workup allows for benchmarking insights into the performance (accuracy and timing) of different simulation techniques, algorithms, and codes. More importantly, the data sets can also be mined to generate data-derived machine learning prediction models that can serve as surrogates for the underlying physics-based models as discussed above. The notebook platform also provides plotting functionality and access to other graphic representations. An example of this strategy is shown in the Fig. 8, where ChemML results for a list of SMILES are compared in a plot.



```
[1]: import openchemistry as oc
     import matplotlib.pyplot as plt

[2]: structure_smiles = ['CN1CNCN(C1)c1ccc(c2c1cccc2)C1NCNCN1', 'SC1CSCC(S1)(S)c1cccnc1', 'c1ccc(nc1)c1scc(c1)c1ccnc(c1)c1ccc2c(c1)cccc2']
     mols = []
     for smiles in structure_smiles:
         mols.append(oc.import_structure(smiles, gen3d=False))

[3]: image_name = 'openchemistry/chemml:latest'
     input_parameters = {}

[4]: results = oc.run_calculations(mols, image_name, input_parameters)

[5]: x = []
     y = []
     for result in results:
         x.append(result.data()['properties']['polarizability'])
         y.append(result.data()['properties']['refractiveIndex'])

[6]: fig, ax = plt.subplots()
     ax.plot(x, y, 'ro')
     ax.set(xlabel='polarizability', ylabel='refractive index', title='Machine learning...')
     ax.grid()
```

Figure 8: A batch of results may be obtained by running multiple calculations in one task, as is done here with ChemML. 3D coordinates are not generated in this example.

Reproducibility in Quantum Chemistry

Scientific teams and organizations should embrace reproducible workflows where all data can be exported, shared, and peer reviewed. For computational chemistry, data should include the full flow of data from initial structures, through to final coordinates, energies, and software versions/binary environments used. The Jupyter and JupyterLab projects have many of the components needed, such as an extensible programming interface, visualization and analysis of data in a common format but it lacks specific workflows for quantum chemistry. This project adds those, and couples notebooks with a data server, and uses an extensible data format definition for static export suitable for long-term archiving of results.

The target audience ranges from a quantum chemistry code developer publishing new methods where this platform enables them to show the input, execution and output of a development snapshot through to end users running calculations on production code. The ability to specify the organization, container name, and version offers the ability to use known versions of codes, and even rerun when fixes are made. It also enhances the peer-review and publication process by offering a full record of what was done computationally, along with the results obtained and a recipe to replicate. As a community we must move towards the routine publication of all of these steps, and consider data standards along with software platforms to reduce the upfront costs of doing so.

The Space Telescope (*spacetelescope/notebooks*, n.d.) selected Jupyter as the primary analysis platform for many of the same reasons it has been used in this project. Fields of scientific research must converge on shared platforms where reproducibility is built in, and share the cost of improving them with customization for each field where it makes sense. The platform described can interface with public databases such as PubChem and QCArchive (Smith et al., 2020) to import existing data, and produce new data with structures suitable for wider dissemination. The data and metadata standards discussed seek to embrace federated storage of data, embracing the goals of FAIR data(*GO FAIR*, n.d.) to make all data produced more discoverable. The use of established open standards such as InChI, InChI key and SMILES link data produced in individual instances to the global data commons with minimal ambiguity.

One of the primary challenges in computational chemistry is to develop a software infrastructure capable of executing codes reproducibly such that others can look at every aspect of what you did, and build upon it. Even quite subtle details can hamper this, such as tolerances used in convergence criterion or software library versions. The system described in this paper uses Docker, along with conversion to other software containers, to package a code along with all of its dependencies. The driver scripts within these containers use Python to drive the execution of the code from JSON input, and to convert the output. This is then layered within an execution framework, database, and user management/file management system. Together they can be deployed from tagged container versions in order to reproducibly create the same environment.

All of this infrastructure is not without cost, and so when considering the publication of data and its more permanent dissemination it is important to consider simpler approaches. This led to the development of integration with Binder, and the use of static repositories with exported data. The open specification of the formats, and the open source Python modules mean that anyone can access and process the data with a rather minimal Jupyter deployment. Even without any of the software, the data specification coupled with input specification offers starting points to manually run calculations with the Docker containers locally. Standalone web widgets enable viewing of processed molecular orbitals without any processing component, and could be extended further in the future to do more within the web browser's JavaScript environment.

Conclusions and Outlook

The development of computational chemistry program packages with its heavy reliance on legacy code has so far rarely included the infrastructure to develop a data standard suitable for sharing data with the wider community, input specification, workflow transparency and reproducibility, as well as analysis and visualization of the outputs. Such an infrastructure should generally be a shared resource, and one that is open so that it can live beyond a single project, company or research group. It is necessary for the developers of the different parts of the solution to communicate more effectively, and enhance the pathways for getting data in and out of each step. Ultimately, the entire workflow needs to use appropriate data standards, and feature validation/peer review to ensure that each code agrees on the form, units, and conventions used.

In this paper, we have demonstrated the utility and potential of a web-based end-to-end software platform to enable automation, transparency, and data-centric research in quantum chemistry along these very lines. The software platform, data, and metadata described have been developed, released, and disseminated under permissive OSI-approved open source licenses. The embrace of open development practices allows this work to be extended in future projects and use cases. The codes feature a modular design to optimize the ease with which its elements can be reused. We have engaged the community – and will continue to do so – to help develop standards, workflows, and working solutions that showcase what can be achieved by combining modern web browser and data server technology. This is at the core of the projects described, and underpins the necessary improvements to better disseminate research. The use of web standards and technologies makes the research more accessible, moving beyond static images to interactive web visualization, web platforms, and snapshots of data that can be interactively queried built on community efforts such as Binder.

As a community it is important to embrace open source, open data, open standards, and open access to reproducible research.

Acknowledgements

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Chicago Operations Office, under Award Number DE-SC0017193. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

MDH was affiliated with Kitware for the development of the software platform described (Contract No. DE-SC0017193), and made minor corrections to the manuscript after changing affiliation to Brookhaven National Laboratory (Contract No. DE-SC0012704). The development of ChemML is supported by the National Science Foundation (NSF) CAREER program (grant No. OAC-1751161), and the New York State Center of Excellence in Materials Informatics (grants No. CMI-1140384 and CMI-1148092). MH gratefully acknowledges support by Phase-I and Phase-II Software Fellowships (grant No. ACI-1547580-479590) of the NSF Molecular Sciences Software Institute (grant No. ACI-1547580) at Virginia Tech (Krylov et al., 2018; Wilkins-Diehr & Crawford, 2018). This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

References

https://opensource.org/licenses. https://opensource.org/licenses

https://opensource.org/licenses/BSD-3-Clause. https://opensource.org/licenses/BSD-3-Clause

https://www.python.org/. https://www.python.org/

https://github.com/openchemistry. https://github.com/OpenChemistry

https://www.json.org/. https://www.json.org/

https://json-ld.org/. https://json-ld.org/

https://support.hdfgroup.org/HDF5/. https://support.hdfgroup.org/HDF5/

https://msgpack.org/. https://msgpack.org/

http://bsonspec.org/. http://bsonspec.org/

https://www.postgresql.org/docs/9.4/datatype-json.html. https://www.postgresql.org/docs/9.4/datatype-json.html

https://www.mongodb.com/. https://www.mongodb.com

https://www.postgresql.org/. https://www.postgresql.org/

https://en.wikipedia.org/wiki/XYZ_file_format. https://en.wikipedia.org/wiki/XYZ_file_format

The semantics of Chemical Markup Language (CML) for computational chemistry : CompChem.. (2012). *J Cheminform*, *4*, 15.

CML: Evolution and design.. (2011). *J Cheminform*, *3*, 44.

The semantics of Chemical Markup Language (CML): dictionaries and conventions.. (2011). *J Cheminform*, *3*, 43.

From data to analysis: linking NWChem and Avogadro with the syntax and semantics of Chemical Markup Language.. (2013). *J Cheminform*, *5*, 25.

Open chemistry: RESTful web APIs, JSON, NWChem and the modern web application.. (2017). *J Cheminform*, *9*, 55.

Open Babel: An open chemical toolbox. (2011). *Journal of Cheminformatics*, *3*(1). https://doi.org/10.1186/1758-2946-3-33

https://www.rdkit.org/. https://www.rdkit.org/

cclib: A library for package-independent computational chemistry algorithms. (2008). *Journal of Computational Chemistry*, *29*(5), 839–845. https://doi.org/10.1002/jcc.20823

The atomic simulation environment-a Python library for working with atoms.. (2017). *J Phys Condens Matter*, *29*, 273002.

The NumPy Array: A Structure for Efficient Numerical Computation. (2011). *Computing in Science & Engineering*, *13*(2), 22–30. https://doi.org/10.1109/mcse.2011.37

https://eigen.tuxfamily.org/. https://eigen.tuxfamily.org/

https://github.com/OpenChemistry/chemicaljson. https://github.com/OpenChemistry/chemicaljson

https://github.com/MolSSI/QCSchema. https://github.com/MolSSI/QCSchema

Avogadro: an advanced semantic chemical editor, visualization, and analysis platform.. (2012). *J Cheminform*, *4*, 17.

Psi4 1.1: An Open-Source Electronic Structure Program Emphasizing Automation Advanced Libraries, and Interoperability. (2017). *Journal of Chemical Theory and Computation*, *13*(7), 3185–3197. https://doi.org/10.1021/acs.jctc.7b00174

InChI, the IUPAC International Chemical Identifier.. (2015). *J Cheminform*, *7*, 23.

http://opensmiles.org/. http://opensmiles.org/

PubChem 2019 update: improved access to chemical data. (2018). *Nucleic Acids Research*, *47*(D1), D1102–D1109. https://doi.org/10.1093/nar/gky1033

Chemspider: A Platform for Crowdsourced Collaboration to Curate Data Derived From Public Compound Databases. (2011). In *Collaborative Computational Technologies for Biomedical Research* (pp. 363–386). John Wiley & Sons Inc. https://doi.org/10.1002/9781118026038.ch22

Loizides, F., & Scmidt, B. (Eds.). (2016). Jupyter Notebooks ? a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (pp. 87–90). IOS Press. https://eprints.soton.ac.uk/403913/

https://jupyter.org/. https://www.jupyter.org

https://girder.readthedocs.io/. https://girder.readthedocs.io/

https://jupyterlab.readthedocs.io/. https://jupyterlab.readthedocs.io/

https://zenodo.org/. https://zenodo.org/

Binder 2.0 - Reproducible interactive, sharable environments for science at scale. (2018). *Proceedings of the 17th Python in Science Conference*. https://doi.org/10.25080/majora-4af1f417-011

https://mybinder.org/. https://mybinder.org/

https://knowledge.figshare.com/. https://knowledge.figshare.com/

https://github.com/openchemistry/mongochemserver. https://github.com/OpenChemistry/mongochemserver

https://github.com/openchemistry/mongochemdeploy. https://github.com/OpenChemistry/mongochemdeploy

Docker: Lightweight Linux Containers for Consistent Development and Deployment. (2014). *Linux Journal*, *239*.

Singularity: Scientific containers for mobility of compute.. (2017). *PLoS One*, *12*, e0177459.

Shifter: Containers for HPC. (2017). *Journal of Physics: Conference Series*, *898*, 082021. https://doi.org/10.1088/1742-6596/898/8/082021

NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. (2010). *Computer Physics Communications*, *181*(9), 1477–1489. https://doi.org/10.1016/j.cpc.2010.04.018

ChemML: A Machine Learning and Informatics Program Package for the Analysis Mining, and Modeling of Chemical and Materials Data. (2020). *WIREs Comput Mol Sci.*, *e1458*. https://doi.org/10.1002/wcms.1458

Building and deploying a cyberinfrastructure for the data-driven design of chemical systems and the exploration of chemical space. (2018). *Molecular Simulation*, *44*(11), 921–929. https://doi.org/10.1080/08927022.2018.1471692

ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost.. (2017). *Chem Sci*, *8*, 3192–3203.

https://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library. https://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

TorchANI: A Free and Open Source PyTorch-Based Deep Learning Implementation of the ANI Neural Network Potentials.. (2020). *J Chem Inf Model*, *60*, 3408–3415.

https://reactjs.org/. https://reactjs.org/

https://angularjs.org/. https://angularjs.org/

https://vuejs.org/. https://vuejs.org/

https://stenciljs.com/. https://stenciljs.com/

3Dmol.js: molecular visualization with WebGL.. (2015). *Bioinformatics*, *31*, 1322–1324.

https://kitware.github.io/vtk-js/. https://kitware.github.io/vtk-js/index.html

https://blog.kitware.com/open-chemistry-avogadro-electronic-structure-in-jupyterlab/. https://blog.kitware.com/open-chemistry-avogadro-electronic-structure-in-jupyterlab/

https://github.com/openchemistry/jupyter-examples. https://github.com/OpenChemistry/jupyter-examples

https://cactus.nci.nih.gov/chemical/structure. https://cactus.nci.nih.gov/chemical/structure

Special Collection: Computational Chemistry. (2019). *ChemistryOpen*, *8*(7), 814–816. https://doi.org/10.1002/open.201900197

Computation and Experiment: A Powerful Combination to Understand and Predict Reactivities. (2016). *Accounts of Chemical Research*, *49*(6), 1311–1319. https://doi.org/10.1021/acs.accounts.6b00068

Benchmarking Quantum Chemical Methods: Are We Heading in the Right Direction?. (2017). *Angewandte Chemie International Edition*, *56*(37), 11011–11018. https://doi.org/10.1002/anie.201611308

https://webbook.nist.gov/chemistry/. https://webbook.nist.gov/chemistry/

https://github.com/spacetelescope/notebooks. https://github.com/spacetelescope/notebooks

The MolSSI QCArchive Project: An Open-Source Platform to Compute Organize, and Share Quantum Chemistry Data. (2020). https://doi.org/10.26434/chemrxiv.11908356.v1

https://www.go-fair.org/. https://www.go-fair.org/

Perspective: Computational chemistry software and its advancement as illustrated through three grand challenge cases for molecular science.. (2018). *J Chem Phys*, *149*, 180901.

NSF's Inaugural Software Institutes: The Science Gateways Community Institute and the Molecular Sciences Software Institute. (2018). *Computing in Science & Engineering*, *20*(5), 26–38. https://doi.org/10.1109/mcse.2018.05329813