OpenFF Evaluator Documentation

openff-evaluator

Apr 25, 2022

GETTING STARTED

An automated and scalable framework for curating, manipulating, and computing data sets of physical properties from molecular simulation and simulation data.

The framework is built around four central ideas:

- **Flexibility:** New physical properties, data sources and calculation approaches are easily added via an extensible plug-in system and a flexible workflow engine.
- **Automation:** *Physical property measurements* are readily importable from open data sources (such as the NIST ThermoML Archive) through the data set APIs, and automatically calculated using either the built-in or user specified calculation schemas.
- **Scalability:** Calculations are readily scalable from single machines and laptops up to large HPC clusters and supercomputers through seamless integration with libraries such as dask.
- **Efficiency:** Properties are estimated using the fastest approach available to the framework, whether that be through evaluating a trained surrogate model, re-evaluating cached simulation data, or by running simulations directly.

CHAPTER

ONE

CALCULATION APPROACHES

The framework is designed around the idea of allowing multiple calculation approaches for estimating the same set of properties, in addition to estimation directly from molecular simulation, all using a uniform API.

The primary purpose of this is to take advantage of the many techniques exist which are able to leverage data from previous simulations to rapidly estimate sets of properties, such as reweighting cached simulation data, or evaluating surrogate models trained upon cached data. The most rapid approach which may accurately estimate a set of properties is automatically determined by the framework on the fly.

Each approach supported by the framework is implemented as a *calculation layer*. Two such layers are currently supported (although new calculation layers can be readily added via the plug-in system):

- evaluating physical properties directly from molecular simulation using the *SimulationLayer*.
- reprocessing cached simulation data with MBAR reweighting using the *ReweightingLayer*.

CHAPTER

TWO

SUPPORTED PHYSICAL PROPERTIES

The framework has built-in support for evaluating a number of *physical properties*, ranging from relatively 'cheap' to compute properties such as liquid densities, up to more computationally demanding properties such as solvation free energies and host-guest binding affinities.

Included for most of these properties is the ability to calculate their derivatives with respect to force field parameters, making the framework ideal for evaluating an objective function and it's gradient as part of a force field optimisation.

Table 1: The physical properties which are natively supported by the framework.

	Direct Simulation		*MBAR Reweighting*	
	Supported	Gradients	Supported	Gradients
Density	✓	✓	✓	✓
Dielectric Constant	✓	✓*	✓	✓*
$H_{vaporization}$	✓	✓	✓	✓
H_{mixing}	✓	✓	✓*	✓
V_{excess}	✓	✓	✓	✓
$G_{solvation}$	✓	✓*	×	×
$G_{host\text{-}guest}$ (**beta**)	✓*	×	×	×

** Entries marked with an asterisk are supported but have not yet been extensively tested and validated.*

See the *physical properties overview page* for more details.

2.1 Installation

The OpenFF Evaluator is currently installable either through `conda` or directly from the source code. Whichever route is chosen, it is recommended to install the framework within a conda environment and allow the conda package manager to install the required and optional dependencies.

More information about conda and instructions to perform a lightweight miniconda installation can be found here. It will be assumed that these have been followed and conda is available on your machine.

2.1.1 Installation from Conda

To install the `openff-evaluator` from the `conda-forge` channel simply run:

```
conda install -c conda-forge openff-evaluator
```

2.1.2 Recommended Dependencies

If you have access to the fantastic OpenEye toolkit we recommend installing this to enable (among many other things) the use of the `BuildDockedCoordinates` protocol and faster conformer generation / AM1BCC partial charge calculations:

```
conda install -c openeye openeye-toolkits
```

To parameterize systems with the Amber `tleap` tool using a `TLeapForceFieldSource` the `ambertools` package must be installed:

```
conda install -c conda-forge 'ambertools >=19.0'
```

2.1.3 Installation from Source

To install the OpenFF Evaluator from source begin by cloning the repository from github:

```
git clone https://github.com/openforcefield/openff-evaluator.git
cd openff-evaluator
```

Create a custom conda environment which contains the required dependencies and activate it:

```
conda env create --name openff-evaluator --file devtools/conda-envs/test_env.yaml
conda activate openff-evaluator
```

Finally, install the estimator itself:

```
python setup.py develop
```

2.2 Architecture

The openff-evaluator framework is constructed as a collection of modular components, each performing a specific role within the estimation of physical property data sets. These components are designed to be as extensible as possible, with support for user created plug-ins built into their core.

Fig. 1: An overview of the openff-evaluators modular design. The framework is split into a 'client-side' which handles the curation and preparation of data sets, and a 'server-side' which performs the estimation of the data sets.

The framework is implemented as a *client-server* architecture. This design allows users to spin up *Evaluator Server* instances on whichever compute resources they may have available (from a single machine up to a large HPC cluster), and to which *Evaluator Client* objects may connect to both request that data sets be estimated, and to query and retrieve the results of those requests.

The *client-side* of the framework is predominantly responsible for providing APIs and objects for:

- curating *data sets* of physical properties from open data sources.
- specifing custom *calculation schemas* which describe how individual properties should be computed.
- requesting that data sets be estimated by a running *Evaluator Server* instance.
- retrieving the results of estimation requests from a running *Evaluator Server* instance.

while the *server-side* is responsible for:

- receiving estimation requests from an *Evaluator Client* object.
- automatically determining which *calculation approach* to use for each property in the request.
- executing those requests across the available *compute resources* following the calculation schemas provided by the client
- *caching data* from any calculations which may be useful for future calculations.

All communication between servers and clients is handled through the TCP protocol.

2.3 Evaluator Client

The `EvaluatorClient` object is responsible for both submitting requests to estimate a data set of properties to a running *Evaluator Server* instance, and for pulling back the results of those requests when complete.

An `EvaluatorClient` object may optionally be created using a set of `ConnectionOptions` which specifies the network address of the running *Evaluator Server* instance to connect to:

```
# Specify the address of a server running on the local machine.
connection_options = ConnectionOptions(server_address="localhost", server_port=8000)
# Create the client object
evaluator_client = EvaluatorClient(connection_options)
```

2.3.1 Requesting Estimates

The client can request the estimation of a data set of properties using the `request_estimate()` function:

```
# Specify the data set.
data_set = PhysicalPropertyDataSet()
data_set.add_properties(...)

# Specify the force field source.
force_field = SmirnoffForceFieldSource.from_path("openff-1.0.0.offxml")

# Specify some estimation options (optional).
options = client.default_request_options(data_set, force_field)

# Specify the parameters to differentiate with respect to (optional).
gradient_keys = [
    ParameterGradientKey(tag="vdW", smirks="[#6X4:1]", attribute="epsilon")
]

# Request the estimation of the data set.
request, errors = evaluator_client.request_estimate(
    data_set,
    force_field,
    options,
    gradient_keys
)
```

A request must at minimum specify:

- the *data set* of physical properties to estimate.
- the *force field parameters* to estimate the data set using.

and may also optionally specify:

- the *options* to use when estimating the property set.
- the parameters to differentiate each physical property estimate with respect to.

Note: Gradients can currently only be computed for requests using a SMIRNOFF based force field.

The `request_estimate()` function returns back two objects:

- a `Request` object which can be used to retrieve the results of the request and,
- an `EvaluatorException` object which will be populated if any errors occured while submitting the request.

The `Request` object is similar to a `Future` object, in that it is an object which can be used to query the current status of a request either asynchronously:

```
results = request.results(synchronous=False)
```

or synchronously:

```
results = request.results(synchronous=True)
```

The results (which may currently be incomplete) are returned back as a `RequestResult` object.

The `Request` object is fully JSON serializable:

```
# Save the request to JSON
request.json(file_path="request.json", format=True)
# Load the request from JSON
request = Request.from_json(file_path="request.json")
```

making it easy to keep track of any open requests.

2.3.2 Request Options

The `RequestOptions` object allows greater control over how properties are estimated by the server. It currently allows control over:

- `calculation_layers`: The *calculation layers* which the server should attempt to use when estimating the data set. The order which the layers are specified in this list is the order which the server will attempt to use each layer.
- `calculation_schemas`: The *calculation schemas* to use for each allowed calculation layer per class of property. These will be automatically populated in the cases where no user specified schema is provided, and where a default schema has been registered with the plugin system for the particular layer and property type.

If no options are passed to `request_estimate()` a default set will be generated through a call to `default_request_options()`. For more information about how default calculation schemas are registered, see the *Default Schemas* section.

2.3.3 Force Field Sources

Different force field representations (e.g. `SMIRNOFF`, `TLeap`, `LigParGen`) are defined within the framework as `ForceFieldSource` objects. A force field source should specify *all* of the options which would be required by a particular force field, such as the non-bonded cutoff or the charge scheme if not specified directly in the force field itself.

Currently the framework has built in support for force fields applied via:

- the OpenFF toolkit (`SmirnoffForceFieldSource`).
- the `tleap` program from the AmberTools suite (`LigParGenForceFieldSource`).
- an instance of the LigParGen server (`LigParGenForceFieldSource`).

The client will automatically adapt any of the built-in calculation schemas which are based off of the `WorkflowCalculationSchema` to use the correct workflow protocol (`BuildSmirnoffSystem`, `BuildTLeapSystem` or `BuildLigParGenSystem`) for the requested force field.

2.4 Evaluator Server

The `EvaluatorServer` object is responsible for coordinating the estimation of physical property data sets as requested by *evaluator clients*. Its primary responsibilities are to:

- recieve incoming requests from an *evaluator clients* to either estimate a dataset of properties, or to query the status of a previous request.
- request that each specified *calculation layers* attempt to estimate the data set of properties, cascading unestimated properties through the different layers.

An `EvaluatorServer` must be created with an accompanying *calculation backend* which will be responsible for distributing any calculations launched by the different calculation layers:

```
with DaskLocalCluster() as calculation_backend:

    evaluator_server = EvaluatorServer(calculation_backend)
    evaluator_server.start()
```

It may also be optionally created using a specific *storage backend* if the default `LocalFileStorage` is not sufficient:

```
with DaskLocalCluster() as calculation_backend:

    storage_backend = LocalFileStorage()

    evaluator_server = EvaluatorServer(calculation_backend, storage_backend)
    evaluator_server.start()
```

By default the server will run synchronously until it is killed, however it may also be run asynchronously such that it can be interacted with directly by a client in the same script:

```
with DaskLocalCluster() as calculation_backend:

    with EvaluatorServer(calculation_backend) as evaluator_server:

        # Specify the data set.
        data_set = PhysicalPropertyDataSet()
        data_set.add_properties(...)

        # Specify the force field source.
        force_field = SmirnoffForceFieldSource.from_path("openff-1.0.0.offxml")

        # Request the estimation of the data set.
        request, errors = evaluator_client.request_estimate(data_set,force_field)
        # Wait for the results.
        results = request.results(synchronous=True)
```

2.4.1 Estimation Batches

When a server recieves a request from a client, it will attempt to split the requested set of properties into smaller batches, represented by the `Batch` object. The server is currently only able to mark entire batches of estimated properties as being completed, as opposed to individual properties.

Currently the server supports two ways of batching properties:

- `SameComponents`: All properties measured for the substance containing the *same* components will be batched together. As an example, the density of a 80:20 and a 20:80 mix of ethanol and water would be batched together, but the density of pure ethanol and the density of pure water would be placed into separate batches.
- `SharedComponents`: All properties measured for substances containing at least one common component will be batched together. As an example, the densities of 80:20 and 20:80 mixtures of ethanol and water, and the pure densities of ethanol and water would be batched together.

The mode of batching is set by the client using the `batch_mode` attribute of the request options.

2.5 Tutorial 01 - Loading Data Sets

In this tutorial we will be exploring the frameworks utilities for loading and manipulating data sets of physical property measurements. The tutorial will cover

- Loading a data set of density measurements from NISTs ThermoML Archive
- Filtering the data set down using a range of criteria, including temperature pressure, and composition.
- Supplementing the data set with enthalpy of vaporization (ΔH_v) data sourced directly from the literature

If you haven't yet installed the OpenFF Evaluator framework on your machine, check out the *installation instructions here!*

Note: If you are running this tutorial in google colab you will need to run a setup script instead of following the installation instructions:

[1]:
```
# !wget https://raw.githubusercontent.com/openforcefield/openff-evaluator/master/docs/
↪tutorials/colab_setup.ipynb
# %run colab_setup.ipynb
```

For the sake of clarity all warnings will be disabled in this tutorial:

[2]:
```
import warnings
warnings.filterwarnings('ignore')
import logging
logging.getLogger("openff.toolkit").setLevel(logging.ERROR)
```

2.5.1 Extracting Data from ThermoML

For anyone who is not familiar with the ThermoML archive - it is a fantastic database of physical property measurements which have been extracted from data published in the

- Journal of Chemical and Engineering Data
- Journal of Chemical Thermodynamics
- Fluid Phase Equilibria
- Thermochimica Acta
- International Journal of Thermophysics

journals. It includes data for a wealth of different physical properties, from simple densities and melting points, to activity coefficients and osmotic coefficients, all of which is freely available. As such, it serves as a fantastic resource for benchmarking and optimising molecular force fields against.

The Evaluator framework has built-in support for extracting this wealth of data, storing the data in easy to manipulate python objects, and for automatically re-computing those properties using an array of calculation techniques, such as molecular simulations and, in future, from trained surrogate models.

This support is provided by the `ThermoMLDataSet` object:

[3]:
```
from openff.evaluator.datasets.thermoml import ThermoMLDataSet
```

The `ThermoMLDataSet` object offers two main routes for extracting data the the archive:

- extracting data directly from the NIST ThermoML web server

- extracting data from a local ThermoML XML archive file

Here we will be extracting data directly from the web server. To pull data from the web server we need to specifiy the digital object identifiers (DOIs) of the data we wish to extract - these correspond to the DOI of the publication that the data was initially sourced from.

For this tutorial we will be extracting data using the following DOIs:

[4]:
```
data_set = ThermoMLDataSet.from_doi(
    "10.1016/j.fluid.2013.10.034",
    "10.1021/je1013476",
)
```

We can inspect the data set to see how many properties were loaded:

[5]:
```
len(data_set)
```

[5]:
```
275
```

and for how many different substances those properties were measured for:

[6]:
```
len(data_set.substances)
```

[6]:
```
254
```

We can also easily check which types of properties were loaded in:

[7]:
```
print(data_set.property_types)
```

```
{'EnthalpyOfMixing', 'Density'}
```

2.5.2 Filtering the Data Set

The data set object we just created contains many different functions which will allow us to filter the data down, retaining only those measurements which are of interest to us.

The first thing we will do is filter out all of the measurements which aren't density measurements:

[8]:
```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByPropertyTypes,
    FilterByPropertyTypesSchema
)

data_set = FilterByPropertyTypes.apply(
    data_set, FilterByPropertyTypesSchema(property_types=["Density"])
)

print(data_set.property_types)
```

```
{'Density'}
```

Next we will filter out all measurements which were made away from atmospheric conditions:

[9]:
```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByPressure,
    FilterByPressureSchema,
```

(continues on next page)

(continued from previous page)

```
    FilterByTemperature,
    FilterByTemperatureSchema,
)

print(f"There were {len(data_set)} properties before filtering")

# First filter by temperature.
data_set = FilterByTemperature.apply(
    data_set,
    FilterByTemperatureSchema(minimum_temperature=298.0, maximum_temperature=298.2)
)
# and then by pressure
data_set = FilterByPressure.apply(
    data_set,
    FilterByPressureSchema(minimum_pressure=101.224, maximum_pressure=101.426)
)

print(f"There are now {len(data_set)} properties after filtering")
```

```
There were 213 properties before filtering
There are now 9 properties after filtering
```

Finally, we will filter out all measurements which were not measured for either ethanol (CCO) or isopropanol (CC(C)O):

[10]:
```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterBySmiles,
    FilterBySmilesSchema,
)

data_set = FilterBySmiles.apply(
    data_set,
    FilterBySmilesSchema(smiles_to_include=["CCO", "CC(C)O"])
)

print(f"There are now {len(data_set)} properties after filtering")
```

```
There are now 2 properties after filtering
```

We will convert the filtered data to a pandas `DataFrame` to more easily visualize the final data set:

[11]:
```
pandas_data_set = data_set.to_pandas()
pandas_data_set[
    ["Temperature (K)", "Pressure (kPa)", "Component 1", "Density Value (g / ml)",
↪"Source"]
].head()
```

[11]:

	Temperature (K)	Pressure (kPa)	Component 1	Density Value (g / ml)	Source
0	298.15	101.325	CC(C)O	0.78270	10.1016/j.fluid.2013.10.034
1	298.15	101.325	CCO	0.78507	10.1021/je1013476

Through filtering, we have now cut down from over 250 property measurements down to just 2. There are many more

possible filters which can be applied. All of these and more information about the data set object can be found in the `PhysicalPropertyDataSet` (from which the `ThermoMLDataSet` class inherits) API documentation.

2.5.3 Adding Extra Data

For the final part of this tutorial, we will be supplementing our newly filtered data set with some enthalpy of vaporization (ΔH_v) measurements sourced directly from the literature (as opposed to from the ThermoML archive).

We will be sourcing values of the ΔH_v of ethanol and isopropanol, summarised in the table below, from the Enthalpies of vaporization of some aliphatic alcohols publication:

Compound	Temperature / K	ΔH_v / $kJmol^{-1}$	$\delta\Delta H_v$ / $kJmol^{-1}$
Ethanol	298.15	42.26	0.02
Isopropanol	298.15	45.34	0.02

In order to create a new ΔH_v measurements, we will first define the state (namely temperature and pressure) that the measurements were recorded at:

[12]:
```
from openff.units import unit
from openff.evaluator.thermodynamics import ThermodynamicState

thermodynamic_state = ThermodynamicState(
    temperature=298.15 * unit.kelvin, pressure=1.0 * unit.atmosphere
)
```

Note: Here we have made use of the ``openff.evaluator.unit`` module to attach units to the temperatures and pressures we are filtering by. This module simply exposes a ``UnitRegistry`` from the fantasticpintlibrary. Pint provides full support for attaching to units to values and is used extensively throughout this framework.

the substances that the measurements were recorded for:

[13]:
```
from openff.evaluator.substances import Substance

ethanol = Substance.from_components("CCO")
isopropanol = Substance.from_components("CC(C)O")
```

and the source of this measurement (defined as the DOI of the publication):

[14]:
```
from openff.evaluator.datasets import MeasurementSource

source = MeasurementSource(doi="10.1016/S0021-9614(71)80108-8")
```

We will combine this information with the values of the measurements to create an object which encodes each of the ΔH_v measurements

[15]:
```
from openff.evaluator.datasets import PropertyPhase
from openff.evaluator.properties import EnthalpyOfVaporization

ethanol_hvap = EnthalpyOfVaporization(
    thermodynamic_state=thermodynamic_state,
    phase=PropertyPhase.Liquid | PropertyPhase.Gas,
    substance=ethanol,
    value=42.26*unit.kilojoule / unit.mole,
```

(continues on next page)

(continued from previous page)

```
    uncertainty=0.02*unit.kilojoule / unit.mole,
    source=source
)
isopropanol_hvap = EnthalpyOfVaporization(
    thermodynamic_state=thermodynamic_state,
    phase=PropertyPhase.Liquid | PropertyPhase.Gas,
    substance=isopropanol,
    value=45.34*unit.kilojoule / unit.mole,
    uncertainty=0.02*unit.kilojoule / unit.mole,
    source=source
)
```

These properties can then be added to our data set:

[16]:
```
data_set.add_properties(ethanol_hvap, isopropanol_hvap)
```

If we print the data set again using pandas we should see that our new measurements have been added:

[17]:
```
pandas_data_set = data_set.to_pandas()
pandas_data_set[
    ["Temperature (K)",
     "Pressure (kPa)",
     "Component 1",
     "Density Value (g / ml)",
     "EnthalpyOfVaporization Value (kJ / mol)",
     "Source"
     ]
].head()
```

[17]:

	Temperature (K)	Pressure (kPa)	Component 1	Density Value (g / ml)	EnthalpyOfVaporization Value (kJ / mol)	Source
0	298.15	101.325	CC(C)O	0.78270	NaN	10.1016/j.fluid.2013.10.034
1	298.15	101.325	CCO	0.78507	NaN	10.1021/je1013476
2	298.15	101.325	CCO	NaN	42.26	10.1016/S0021-9614(71)80108-8
3	298.15	101.325	CC(C)O	NaN	45.34	10.1016/S0021-9614(71)80108-8

2.5.4 Conclusion

We will finish off this tutorial by saving the data set we have created as a JSON file for future use:

[18]:
```
data_set.json("filtered_data_set.json", format=True);
```

And that concludes the first tutorial. For more information about data sets in the Evaluator framework check out the *data set* and *ThermoML* documentation.

In the next tutorial we will be estimating the data set we have created here using molecular simulation.

If you have any questions and / or feedback, please open an issue on the GitHub issue tracker.

2.6 Tutorial 02 - Estimating Data Sets

In this tutorial we will be estimating the data set we created in the *first tutorial* using molecular simulation. The tutorial will cover:

- loading in the data set to estimate, and the force field parameters to use in the calculations.
- defining custom calculation schemas for the properties in our data set.
- estimating the data set of properties using an *Evaluator server* instance.
- retrieving the results from the server and storing them on disk.

Note: If you are running this tutorial in google colab you will need to run a setup script instead of following the installation instructions:

[1]:
```
# !wget https://raw.githubusercontent.com/openforcefield/openff-evaluator/master/docs/
↪tutorials/colab_setup.ipynb
# %run colab_setup.ipynb
```

For this tutorial make sure that you are using a GPU accelerated runtime.

For the sake of clarity all warnings will be disabled in this tutorial:

[2]:
```
import warnings
warnings.filterwarnings('ignore')
import logging
logging.getLogger("openforcefield").setLevel(logging.ERROR)
```

We will also enable time-stamped logging to help track the progress of our calculations:

[3]:
```
from openff.evaluator.utils import setup_timestamp_logging
setup_timestamp_logging()
```

2.6.1 Loading the Data Set and Force Field Parameters

We will begin by loading in the data set which we created in the previous tutorial:

[4]:
```
from openff.evaluator.datasets import PhysicalPropertyDataSet

data_set_path = "filtered_data_set.json"

# If you have not yet completed that tutorial or do not have the data set file
# available, a copy is provided by the framework:

# from openff.evaluator.utils import get_data_filename
# data_set_path = get_data_filename("tutorials/tutorial01/filtered_data_set.json")

data_set = PhysicalPropertyDataSet.from_json(data_set_path)
```

As a reminder, this data contains the experimentally measured density and H_{vap} measurements for ethanol and isopropanol at ambient conditions:

[5]:
```
data_set.to_pandas().head()
```

[5]:
```
   Temperature (K)  ...                          Source
0           298.15  ...    10.1016/j.fluid.2013.10.034
1           298.15  ...              10.1021/je1013476
2           298.15  ...  10.1016/S0021-9614(71)80108-8
3           298.15  ...  10.1016/S0021-9614(71)80108-8

[4 rows x 13 columns]
```

We will also define the set of force field parameters which we wish to use to estimate this data set of properties. The framework has support for estimating force field parameters from a range of sources, including those in the OpenFF SMIRNOFF format, those which can be applied by AmberTools, *and more*.

Each source of a force field has a corresponding source object in the framework. In this tutorial we will be using the OpenFF Parsley force field which is based off of the SMIRNOFF format:

[6]:
```
from openff.evaluator.forcefield import SmirnoffForceFieldSource

force_field_path = "openff-1.0.0.offxml"
force_field_source = SmirnoffForceFieldSource.from_path(force_field_path)
```

2.6.2 Defining the Calculation Schemas

The next step we will take will be to define a custom calculation schema for each type of property in our data set.

A calculation schema is the blueprint for how a type of property should be calculated using a particular *calculation approach*, such as directly by simulation, by reprocessing cached simulation data or, in future, a range of other options.

The framework has built-in schemas defining how densities and H_{vap} should be estimated from molecular simulation, covering all aspects from coordinate generation, force field assignment, energy minimisation, equilibration and finally the production simulation and data analysis. All of this functionality is implemented via the frameworks built-in, lightweight *workflow engine*, however we won't dive into the details of this until a later tutorial.

For the purpose of this tutorial, we will simply modify the default calculation schemas to reduce the number of molecules to include in our simulations to speed up the calculations. This step can be skipped entirely if the default options (which we recommend using for 'real-world' calculations) are to be used:

[7]:
```
from openff.evaluator.properties import Density, EnthalpyOfVaporization

density_schema = Density.default_simulation_schema(n_molecules=256)
h_vap_schema = EnthalpyOfVaporization.default_simulation_schema(n_molecules=256)
```

We could further use this method to set either the absolute or the relative uncertainty that the property should be estimated to within. If either of these are set, the simulations will automatically be extended until the target uncertainty in the property has been met.

For our purposes however we won't set any targets, leaving the simulations to run for the default 1 ns.

To use these custom schemas, we need to add them to the a request options object which defines all of the options for estimating our data set:

[8]:
```
from openff.evaluator.client import RequestOptions

# Create an options object which defines how the data set should be estimated.
```

(continues on next page)

(continued from previous page)

```
estimation_options = RequestOptions()
# Specify that we only wish to use molecular simulation to estimate the data set.
estimation_options.calculation_layers = ["SimulationLayer"]

# Add our custom schemas, specifying that the should be used by the 'SimulationLayer'
estimation_options.add_schema("SimulationLayer", "Density", density_schema)
estimation_options.add_schema("SimulationLayer", "EnthalpyOfVaporization", h_vap_schema)
```

2.6.3 Launching the Server

The framework is split into two main applications - an `EvaluatorServer` and an `EvaluatorClient`.

The `EvaluatorServer` is the main object which will perform any and all calculations needed to estimate sets of properties. It is design to run on whichever compute resources you may have available (whether that be a single machine or a high performance cluster), wait until a user requests a set of properties be estimated, and then handle that request.

The `EvaluatorClient` is the object used by the user to send requests to estimate data sets to running server instances over a TCP connection. It is also used to query the server to see when that request has been fulfilled, and to pull back any results.

Let us begin by spawning a new server instance.

To launch a server, we need to define how this object is going to interact with the compute resource it is running on.

This is accomplished using a *calculation backend*. While there are several to choose from depending on your needs, well will go with a simple `dask` based one designed to run on a single machine:

[9]:
```
from openff.evaluator.backends import ComputeResources
from openff.evaluator.backends.dask import DaskLocalCluster

calculation_backend = DaskLocalCluster(
    number_of_workers=1,
    resources_per_worker=ComputeResources(
        number_of_threads=1,
        number_of_gpus=1,
        preferred_gpu_toolkit=ComputeResources.GPUToolkit.CUDA
    ),
)
calculation_backend.start()
```

Here we have specified that we want to run our calculations on a single worker which has access to a single GPU.

With that defined, we can go ahead and spin up the server:

[10]:
```
from openff.evaluator.server import EvaluatorServer

evaluator_server = EvaluatorServer(calculation_backend=calculation_backend)
evaluator_server.start(asynchronous=True)
```

```
02:47:53.961 INFO     Server listening at port 8000
```

The server will run asynchronously in the background waiting until a client connects and requests that a data set be estimated.

2.6.4 Estimating the Data Set

With the server spun up we can go ahead and connect to it using an `EvaluatorClient` and request that it estimate our data set using the custom options we defined earlier:

[11]:
```
from openff.evaluator.client import EvaluatorClient
evaluator_client = EvaluatorClient()

request, exception = evaluator_client.request_estimate(
    property_set=data_set,
    force_field_source=force_field_source,
    options=estimation_options,
)

assert exception is None
```

```
02:47:54.012 INFO     Received estimation request from ('127.0.0.1', 50618)
```

The server will now receive the requests and begin whirring away fulfilling it. It should be noted that the `request_estimate()` function returns two values - a `request` object, and an `exception` object. If all went well (as it should do here) the `exception` object will be `None`.

The `request` object represents the request which we just sent to the server. It stores the unique id which the server assigned to the request, as well as the address of the server that the request was sent to.

The `request` object is primarily used to query the current state of our request, and to pull down the results when it the request finishes. Here we will use it it synchronously query the server every 30 seconds until our request has completed.

[12]:
```
# Wait for the results.
results, exception = request.results(synchronous=True, polling_interval=30)
assert exception is None
```

Note: we could also asynchronously query for the results of the request. The resultant results object would then contain the partial results of any completed estimates, as well as any exceptions raised during the estimation.

2.6.5 Inspecting the Results

Now that the server has finished estimating our data set and returned the results to us, we can begin to inspect the results of the calculations:

[13]:
```
print(len(results.queued_properties))

print(len(results.estimated_properties))

print(len(results.unsuccessful_properties))
print(len(results.exceptions))
```

```
0
4
0
0
```

We can (hopefully) see here that there were no exceptions raised during the calculation, and that all of our properties were successfully estimated.

We will extract the estimated data set and save this to disk:

[14]:
```
results.estimated_properties.json("estimated_data_set.json", format=True);
```

2.6.6 Conclusion

And that concludes the second tutorial. In the next tutorial we will be performing some basic analysis on our estimated results.

If you have any questions and / or feedback, please open an issue on the GitHub issue tracker.

2.7 Tutorial 03 - Analysing Data Sets

In this tutorial we will be analysing the results of the calculations which we performed in the *second tutorial*. The tutorial will cover:

- comparing the estimated data set with the experimental data set.
- plotting the two data sets.

Note: If you are running this tutorial in google colab you will need to run a setup script instead of following the installation instructions:

[1]:
```
# !wget https://raw.githubusercontent.com/openforcefield/openff-evaluator/master/docs/
↪tutorials/colab_setup.ipynb
# %run colab_setup.ipynb
```

For the sake of clarity all warnings will be disabled in this tutorial:

[2]:
```
import warnings
warnings.filterwarnings('ignore')
import logging
logging.getLogger("openforcefield").setLevel(logging.ERROR)
```

2.7.1 Loading the Data Sets

We will begin by loading both the experimental data set and the estimated data set:

[3]:
```
from openff.evaluator.datasets import PhysicalPropertyDataSet

experimental_data_set_path = "filtered_data_set.json"
estimated_data_set_path = "estimated_data_set.json"

# If you have not yet completed the previous tutorials or do not have the data set files
# available, copies are provided by the framework:

# from openff.evaluator.utils import get_data_filename
# experimental_data_set_path = get_data_filename(
#     "tutorials/tutorial01/filtered_data_set.json"
# )
# estimated_data_set_path = get_data_filename(
```

(continues on next page)

(continued from previous page)

```
#     "tutorials/tutorial02/estimated_data_set.json"
# )

experimental_data_set = PhysicalPropertyDataSet.from_json(experimental_data_set_path)
estimated_data_set = PhysicalPropertyDataSet.from_json(estimated_data_set_path)
```

if everything went well from the previous tutorials, these data sets will contain the density and H_{vap} of ethanol and isopropanol:

[4]:
```
experimental_data_set.to_pandas().head()
```

[4]:
```
   Temperature (K)  ...                          Source
0           298.15  ...    10.1016/j.fluid.2013.10.034
1           298.15  ...              10.1021/je1013476
2           298.15  ...  10.1016/S0021-9614(71)80108-8
3           298.15  ...  10.1016/S0021-9614(71)80108-8

[4 rows x 13 columns]
```

[5]:
```
estimated_data_set.to_pandas().head()
```

[5]:
```
   Temperature (K)  ...           Source
0           298.15  ...  SimulationLayer
1           298.15  ...  SimulationLayer
2           298.15  ...  SimulationLayer
3           298.15  ...  SimulationLayer

[4 rows x 13 columns]
```

2.7.2 Extracting the Results

We will now compare how the value of each property estimated by simulation deviates from the experimental measurement.

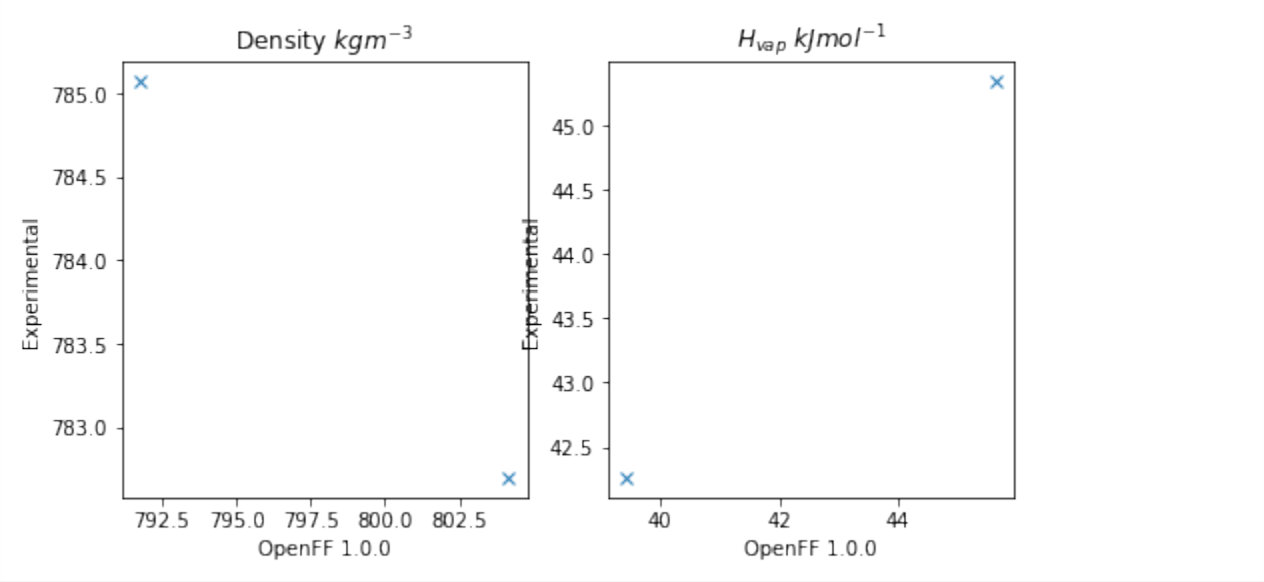
To do this we will extract a list which contains pairs of experimental and evaluated properties. We can easily match properties based on the unique ids which were automatically assigned to them on their creation:

[6]:
```
properties_by_type = {
    "Density": [],
    "EnthalpyOfVaporization": []
}

for experimental_property in experimental_data_set:

    # Find the estimated property which has the same id as the
    # experimental property.
    estimated_property = next(
        x for x in estimated_data_set if x.id == experimental_property.id
    )

    # Add this pair of properties to the list of pairs
    property_type = experimental_property.__class__.__name__
    properties_by_type[property_type].append((experimental_property, estimated_property))
```

2.7.3 Plotting the Results

We will now compare the experimental results to the estimated ones by plotting them using `matplotlib`:

[7]:
```
from matplotlib import pyplot

# Create the figure we will plot to.
figure, axes = pyplot.subplots(nrows=1, ncols=2, figsize=(8.0, 4.0))

# Set the axis titles
axes[0].set_xlabel('OpenFF 1.0.0')
axes[0].set_ylabel('Experimental')
axes[0].set_title('Density $kg m^{-3}$')

axes[1].set_xlabel('OpenFF 1.0.0')
axes[1].set_ylabel('Experimental')
axes[1].set_title('$H_{vap}$ $kJ mol^{-1}$')

# Define the preferred units of the properties
from openff.units import unit

preferred_units = {
    "Density": unit.kilogram / unit.meter ** 3,
    "EnthalpyOfVaporization": unit.kilojoule / unit.mole
}

for index, property_type in enumerate(properties_by_type):

    experimental_values = []
    estimated_values = []

    preferred_unit = preferred_units[property_type]

    # Convert the values of our properties to the preferred units.
    for experimental_property, estimated_property in properties_by_type[property_type]:

        experimental_values.append(
            experimental_property.value.to(preferred_unit).magnitude
        )
        estimated_values.append(
            estimated_property.value.to(preferred_unit).magnitude
        )

    axes[index].plot(
        estimated_values, experimental_values, marker='x', linestyle='None'
    )
```



2.7.4 Conclusion

And that concludes the third tutorial!

If you have any questions and / or feedback, please open an issue on the GitHub issue tracker.

2.8 Tutorial 04 - Optimizing Force Fields

In this tutorial we will be using the OpenFF Evaluator framework in combination with the fantastic ForceBalance software to optimize a molecular force field against the physical property data set we created in the *first tutorial*.

ForceBalance offers a suite of tools for optimizing molecular force fields against a set of target data. Perhaps one of the most fundamental targets to fit against is experimental physical property data. Physical property data has been used extensively for decades to inform the values of non-bonded Van der Waals (VdW) interaction parameters (often referred to as Lennard-Jones parameters).

ForceBalance is seamlessly integrated with the evaluator framework, using it to evaluate the deviations between target experimentally measured data points and those evaluated using the force field being optimized (as well as the gradient of those deviations with respect to the force field parameters being optimized).

The tutorial will cover:

- setting up the input files and directory structure required by ForceBalace.
- setting up an `EvaluatorServer` for *ForceBalance* to connect to.
- running *ForceBalance* using those input files.
- extracting and plotting a number of statistics output during the optimization.

Note: If you are running this tutorial in google colab you will need to run a setup script instead of following the installation instructions:

[1]:
```
# !wget https://raw.githubusercontent.com/openforcefield/openff-evaluator/master/docs/
↪tutorials/colab_setup.ipynb
# %run colab_setup.ipynb
```

For this tutorial make sure that you are using a GPU accelerated runtime.

For the sake of clarity all warnings will be disabled in this tutorial:

[2]:
```
import warnings
warnings.filterwarnings('ignore')
import logging
logging.getLogger("openforcefield").setLevel(logging.ERROR)
```

We will also enable time-stamped logging to help track the progress of our calculations:

[3]:
```
from openff.evaluator.utils import setup_timestamp_logging
setup_timestamp_logging()
```

2.8.1 Setting up the ForceBalance Inputs

In this section we will be creating the directory structure required by *ForceBalance*, and populating it with the required input files.

Creating the Directory Structure

To begin with, we will create a directory to store the starting force field parameters in:

[4]:
```
!mkdir forcefield
```

and one to store the input parameters for our 'fitting target' - in this case a data set of physical properties:

[5]:
```
!mkdir -p targets/pure_data
```

Defining the Training Data Set

With the directories created, we will next specify the data set of physical properties which we will be training the force field against:

[6]:
```
# For convenience we will use the copy shipped with the framework
from openff.evaluator.utils import get_data_filename
data_set_path = get_data_filename("tutorials/tutorial01/filtered_data_set.json")

# Load the data set.
from openff.evaluator.datasets import PhysicalPropertyDataSet
data_set = PhysicalPropertyDataSet.from_json(data_set_path)

# Due to a small bug in ForceBalance we need to zero out any uncertainties
# which are undefined. This will be fixed in future versions.
from openff.evaluator.attributes import UNDEFINED

for physical_property in data_set:
```

(continues on next page)

(continued from previous page)

```
    if physical_property.uncertainty != UNDEFINED:
        continue

    physical_property.uncertainty = 0.0 * physical_property.default_unit()
```

To speed up the runtime of this tutorial, we will only train the force field against measurements made for ethanol

[7]:
```
data_set.filter_by_smiles("CCO")
```

in real optimizations however the data set should be **much** larger than two data points!

With those changes made, we can save the data set in our targets directory:

[8]:
```
# Store the data set in the `pure_data` targets folder:
data_set.json("targets/pure_data/training_set.json");
```

Defining the Starting Force Field Parameters

We will use the OpenFF Parsley 1.0.0 force field as the starting parameters for the optimization. These can be loaded directly into an OpenFF `ForceField` object using the OpenFF toolkit:

[9]:
```
from openforcefield.typing.engines.smirnoff import ForceField
force_field = ForceField('openff-1.0.0.offxml')
```

In order to use these parameters in *ForceBalance*, we need to 'tag' the individual parameters in the force field that we wish to optimize. The toolkit easily enables us to add these tags using cosmetic attributes:

[10]:
```
# Extract the smiles of all unique components in our data set.
from openforcefield.topology import Molecule, Topology

all_smiles = set(
    component.smiles
    for substance in data_set.substances
    for component in substance.components
)

for smiles in all_smiles:

    # Find those VdW parameters which would be applied to those components.
    molecule = Molecule.from_smiles(smiles)
    topology = Topology.from_molecules([molecule])

    labels = force_field.label_molecules(topology)[0]

    # Tag the exercised parameters as to be optimized.
    for parameter in labels["vdW"].values():
        parameter.add_cosmetic_attribute("parameterize", "epsilon, rmin_half")
```

Here we have made use of the toolkit's handy `label_molecules` function to see which VdW parameters will be assigned to the molecules in our data set, and tagged them to be parameterized.

With those tags added, we can save the parameters in the `forcefield` directory:

[11]:
```
# Save the annotated force field file.
force_field.to_file('forcefield/openff-1.0.0-tagged.offxml')
```

*Note: The force field parameters are stored in the*OpenFF SMIRNOFF XML format.

Creating the Main Input File

Next, we will create the main *ForceBalance* input file. For the sake of brevity a default input file which ships with this framework will be used:

[12]:
```
input_file_path = get_data_filename("tutorials/tutorial04/optimize.in")

# Copy the input file into our directory structure
import shutil
shutil.copyfile(input_file_path, "optimize.in")
```

[12]:
```
'optimize.in'
```

While there are many options that can be set within this file, the main options of interest for our purposes appear at the bottom of the file:

[13]:
```
!tail -n 6 optimize.in
```

```
$target
name pure_data
type Evaluator_SMIRNOFF
weight 1.0
openff.evaluator_input options.json
$end
```

Here we have specified that we wish to create a new *ForceBalance* `Evaluator_SMIRNOFF` target called `pure_data` (corresponding to the name of the directory we created in the earlier step).

The main input to this target is the file path to an `options.json` file - it is this file which will specify all the options which should be used when *ForceBalance* requests that our target data set be estimated using the current sets of force field parameters.

We will create this file in the `targets/pure_data` directory later in this section.

The data set is the JSON serialized representation of the `PhysicalPropertyDataSet` we created during the *first tutorial*.

Defining the Estimation Options

The final step before we can start the optimization is to create the set of options which will govern how our data set is estimated using the Evaluator framework.

These options will be stored in an `Evaluator_SMIRNOFF` object:

[14]:
```
from forcebalance.evaluator_io import Evaluator_SMIRNOFF

# Create the ForceBalance options object
target_options = Evaluator_SMIRNOFF.OptionsFile()
# Set the path to the data set
target_options.data_set_path = "training_set.json"
```

This object exposes both a set of *ForceBalance* specific options, as well as the set of Evaluator options.

The *ForceBalance* specific options allow us to define how each type of property will contribute to the optimization objective function (the value which we are trying to minimize):

$$\Delta(\theta) = \sum_{n}^{N} \frac{weight_n}{M_n} \sum_{m}^{M_n} \left(\frac{y_m^{ref} - y_m(\theta)}{denominator_n} \right)^2$$

where N is the number of types of properties (e.g. density, enthalpy of vaporization, etc.), M_n is the number of data points of type n, y_m^{ref} is the experimental value of data point m and $y_m(\theta)$ is the estimated value of data point m using the current force field parameters

In particular, the options object allows us to specify both an amount to scale each type of properties contribution to the objective function by ($weight_n$), and the amount to scale the difference between the experimental and estimated properties ($denominator_n$):

[15]:
```
from openff.units import unit

target_options.weights = {
    "Density": 1.0,
    "EnthalpyOfVaporization": 1.0
}
target_options.denominators = {
    "Density": 30.0 * unit.kilogram / unit.meter ** 3,
    "EnthalpyOfVaporization": 3.0 * unit.kilojoule / unit.mole
}
```

where here we have chosen values that ensure that both types of properties contribute roughly equally to the total objective function.

The Evaluator specific options correspond to a standard `RequestOptions` object:

[16]:
```
from openff.evaluator.client import RequestOptions

# Create the options which evaluator should use.
evaluator_options = RequestOptions()
# Choose which calculation layers to make available.
evaluator_options.calculation_layers = ["SimulationLayer"]

# Reduce the default number of molecules
from openff.evaluator.properties import Density, EnthalpyOfVaporization

density_schema = Density.default_simulation_schema(n_molecules=256)
h_vap_schema = EnthalpyOfVaporization.default_simulation_schema(n_molecules=256)

evaluator_options.add_schema("SimulationLayer", "Density", density_schema)
evaluator_options.add_schema("SimulationLayer", "EnthalpyOfVaporization", h_vap_schema)

target_options.estimation_options = evaluator_options
```

These options allow us to control exactly how each type of property should be estimated, which calculation approaches should be used and more. Here we use the same options are were used in the *second tutorial*

Note: more information about the different estimation options can befound here

And that's the options created! We will finish off by serializing the options into our target directory:

[17]:
```
# Save the options to file.
with open("targets/pure_data/options.json", "w") as file:
    file.write(target_options.to_json())
```

2.8.2 Launching an Evaluator Server

With the *ForceBalance* options created, we can now move onto launching the `EvaluatorServer` which *ForceBalance* will call out to when it needs the data set to be evaluated:

[18]:
```
# Launch the calculation backend which will distribute any calculations.
from openff.evaluator.backends import ComputeResources
from openff.evaluator.backends.dask import DaskLocalCluster

calculation_backend = DaskLocalCluster(
    number_of_workers=1,
    resources_per_worker=ComputeResources(
        number_of_threads=1,
        number_of_gpus=1,
        preferred_gpu_toolkit=ComputeResources.GPUToolkit.CUDA
    ),
)
calculation_backend.start()

# Launch the server object which will listen for estimation requests and schedule any
# required calculations.
from openff.evaluator.server import EvaluatorServer

evaluator_server = EvaluatorServer(calculation_backend=calculation_backend)
evaluator_server.start(asynchronous=True)
```

```
01:30:20.505 INFO     Server listening at port 8000
```

We will not go into the details of this here as this was already covered in the *second tutorial*

2.8.3 Running ForceBalance

With the inputs created and an Evaluator server spun up, we are finally ready to run the optimization! This can be accomplished with a single command:

[19]:
```
!ForceBalance optimize.in
```

If everything went well *ForceBalance* should exit cleanly, and will have stored out newly optimized force field in the `results` directory.

[20]:
```
!ls result/optimize
```

```
openff-1.0.0-tagged_1.offxml  openff-1.0.0-tagged.offxml
```

2.8.4 Plotting the results

As a last step in this tutorial, we will extract the objective function at each iteration from the *ForceBalance* output files and plot this using `matplotlib`.

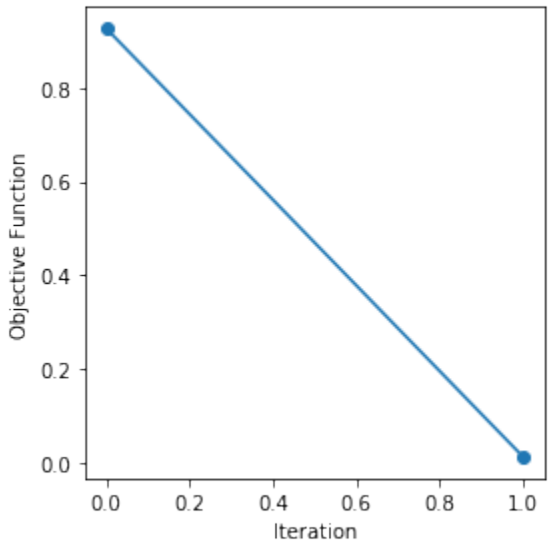
First, we will extract the objective function from the `pickle` serialized output files which can be found in the `optimize.tmp/pure_data/iter_****/` directories:

[21]:
```
from forcebalance.nifty import lp_load

# Determine how many iterations ForceBalance has completed.
from glob import glob
n_iterations = len(glob("optimize.tmp/pure_data/iter*"))

# Extract the objective function at each iteration.
objective_function = []

for iteration in range(n_iterations):

    folder_name = "iter_" + str(iteration).zfill(4)
    file_path = f"optimize.tmp/pure_data/{folder_name}/objective.p"

    statistics = lp_load(file_path)
    objective_function.append(statistics["X"])

print(objective_function)
```

```
[0.9270359101845124, 0.011497456194198362]
```

The objective function is then easily plotted:

[22]:
```
from matplotlib import pyplot
figure, axis = pyplot.subplots(1, 1, figsize=(4, 4))

axis.set_xlabel("Iteration")
axis.set_ylabel("Objective Function")

axis.plot(range(n_iterations), objective_function, marker="o")

figure.tight_layout()
```



2.8.5 Conclusion

And that concludes the fourth tutorial!

If you have any questions and / or feedback, please open an issue on the GitHub issue tracker.

2.9 Property Data Sets

A `PhysicalPropertyDataSet` is a collection of measured physical properties encapsulated as *physical property* objects. They may be created from scratch:

```
# Define a density measurement
density = Density(
    substance=Substance.from_components("O"),
    thermodynamic_state=ThermodynamicState(
        pressure=1.0*unit.atmospheres, temperature=298.15*unit.kelvin
    ),
    phase=PropertyPhase.Liquid,
    value=1.0*unit.gram/unit.millilitre,
    uncertainty=0.0001*unit.gram/unit.millilitre
)

# Add the property to a data set
data_set = PhysicalPropertyDataset()
data_set.add_properties(density)
```

are readily JSON (de)serializable:

```
# Save the data set as a JSON file.
data_set.json(file_path="data_set.json", format=True)
# Load the data set from a JSON file
data_set = PhysicalPropertyDataset.from_json(file_path="data_set.json")
```

and may be converted to pandas `DataFrame` objects:

```
data_set.to_pandas()
```

The framework implements specific data set objects for extracting data measurements directly from a number of open data sources, such as the `ThermoMLDataSet` (see *ThermoML Archive*) which provides utilities for extracting the data from the NIST ThermoML Archive and converting it into the standard framework objects.

Data set objects are directly iterable:

```
for physical_property in data_set:
    ...
```

or can be iterated over for a specific substance:

```
for physical_property in data_set.properties_by_substance(substance):
    ...
```

or for a specific type of property:

```
for physical_property in data_set.properties_by_type("Density"):
    ...
```

2.9.1 Physical Properties

The `PhysicalProperty` object is a base class for any object which describes a measured property of substance, and is defined by a combination of:

- the observed value of the property.
- `Substance` specifying the substance that the measurement was collected for.
- `PropertyPhase` specifying the phase that the measurement was collected in.
- `ThermodynamicState` specifying the thermodynamic conditions under which the measurement was performed

as well as optionally

- the uncertainty in the value of the property.
- a list of `ParameterGradient` which defines the gradient of the property with respect to the model parameters if it was computationally estimated.
- a `Source` specifying the source (either experimental or computational) and provenance of the measurement.

Each type of property supported by the framework, such as a density of an enthalpy of vaporization, must have it's own class representation which inherits from `PhysicalProperty`:

```
# Define a density measurement
density = Density(
    substance=Substance.from_components("O"),
    thermodynamic_state=ThermodynamicState(
        pressure=1.0*unit.atmospheres, temperature=298.15*unit.kelvin
    ),
    phase=PropertyPhase.Liquid,
    value=1.0*unit.gram/unit.millilitre,
    uncertainty=0.0001*unit.gram/unit.millilitre
)
```

2.9.2 Substances

A `Substance` is defined by a number of components (which may have specific roles assigned to them such as being solutes in the system) and the amount of each component in the substance.

To create a pure substance containing only water:

```
water_substance = Substance.from_components("O")
```

To create binary mixture of water and methanol in a 20:80 ratio:

```
binary_mixture = Substance()
binary_mixture.add_component(Component(smiles="O"), MoleFraction(value=0.2))
binary_mixture.add_component(Component(smiles="CO"), MoleFraction(value=0.8))
```

To create a substance of an infinitely dilute paracetamol solute dissolved in water:

```
solution = Substance()
solution.add_component(
    Component(smiles="O", role=Component.Role.Solvent), MoleFraction(value=1.0)
)
solution.add_component(
    Component(smiles="CC(=O)Nc1ccc(O)cc1", role=Component.Role.Solute),␣
↪ExactAmount(value=1)
)
```

2.9.3 Property Phases

The `PropertyPhase` enum describes the possible phases which a measurement was performed in.

While the enum only has three defined phases (`Solid`, `Liquid` and `Gas`), multiple phases can be formed by OR'ing (|) multiple phases together. As an example, to define a phase for a liquid and gas coexisting:

```
liquid_gas_phase = PropertyPhase.Liquid | PropertyPhase.Gas
```

2.9.4 Thermodynamic States

A `ThermodynamicState` specifies a combination of the temperature and (optionally) the pressure at which a measurement is performed:

```
thermodynamic_state = ThermodynamicState(
    temperature=298.15*unit.kelvin, pressure=1.0*unit.atmosphere
)
```

2.10 ThermoML Archive

The `ThermoMLDataSet` object offers an API for extracting physical properties from the NIST ThermoML Archive, both directly from the archive itself or from files stored in the IUPAC- standard ThermoML format.

The API only supports extracting those properties which have been *registered* with the frameworks plug-in system, and does not currently load the full set of metadata available in the archive files.

Note: If the metadata you require is not currently exposed, please open an issue on the GitHub issue tracker to request it.

Currently the framework has built-in support for extracting:

- *Mass density, kg/m3* (`Density`)
- *Excess molar volume, m3/mol* (`ExcessMolarVolume`)
- *Relative permittivity at zero frequency* (`DielectricConstant`)
- *Excess molar enthalpy (molar enthalpy of mixing), kJ/mol* (`EnthalpyOfMixing`)
- *Molar enthalpy of vaporization or sublimation, kJ/mol* (`EnthalpyOfVaporization`)

where here both the ThermoML property name (as defined by the IUPAC XML schema) and the built-in framework class are listed.

2.10.1 Registering Properties

Properties to be extracted from ThermoML archives must have a corresponding class representation to be loading into. This class representation must both:

- inherit from the frameworks `PhysicalProperty` class and
- be registered with the frameworks plug-in system using either the `thermoml_property()` decorator or the `register_thermoml_property()` method.

As an example, a class representation of the ThermoML *'Mass density, kg/m3'* property could be defined and registered with the plug-in system using:

```
@thermoml_property("Mass density, kg/m3", supported_phases=PropertyPhase.Liquid)
class Density(PhysicalProperty):
    """A class representation of a mass density property"""
```

The `thermoml_property()` decorator takes in the name of the ThermoML property (as defined by the IUPAC schema) as well as the phases which the framework will be able to estimate this property in.

Multiple ThermoML properties can be mapped onto a single class using the flexible `register_thermoml_property()` function. For example, the *'Specific volume, m3/kg'* property (which is simply the reciprocal of mass density) may be mapped onto the `Density` object by providing a `conversion_function`:

```
def specific_volume_to_mass_density(specific_volume):
    """Converts a specific volume measurement into a mass
    density.

    Parameters
    ----------
```

(continues on next page)

(continued from previous page)

```
    specific_volume: ThermoMLProperty
        The specific volume measurement to convert.
    """
    mass_density = Density()

    mass_density.value = 1.0 / specific_volume.value

    if mass_density.uncertainty is not None:
        mass_density.uncertainty = 1.0 / mass_density.uncertainty

    mass_density.phase = specific_volume.phase

    mass_density.thermodynamic_state = specific_volume.thermodynamic_state
    mass_density.substance = specific_volume.substance

    return mass_density

# Register the ThermoML property using the conversion function.
register_thermoml_property(
    thermoml_string="Specific volume, m3/kg",
    supported_phases=PropertyPhase.Liquid,
    property_class=Density,
    conversion_function=specific_volume_to_mass_density
)
```

Converting the different density derivatives into a single density class removes the need to produce many very similar class representations of density measurements, and allows a single calculation schema to be defined for all variants.

2.10.2 Loading Data Sets

Data sets are most easily loaded using their digital object identifiers (DOI). For example, to retrieve the ThermoML data set that accompanies this paper, we can simply use the DOI `10.1016/j.jct.2005.03.012`:

```
data_set = ThermoMLDataset.from_doi('10.1016/j.jct.2005.03.012')
```

Data can be pulled from multiple sources at once by specifying multiple identifiers:

```
identifiers = ['10.1021/acs.jced.5b00365', '10.1021/acs.jced.5b00474']
dataset = ThermoMLDataset.from_doi(*identifiers)
```

Entire archives of properties can be downloaded directly from the ThermoML website and parsed by the framework. For example, to create a data set object containing all of the measurements recorded from the International Journal of Thermophysics:

```
# Download the archive of all properties from the IJT journal.
import requests
request = requests.get("https://trc.nist.gov/ThermoML/IJT.tgz", stream=True)

# Make sure the request went ok.
assert request

# Unzip the files into a new 'ijt_files' directory.
```

(continues on next page)

(continued from previous page)

```
import io, tarfile
tar_file = tarfile.open(fileobj=io.BytesIO(request.content))
tar_file.extractall("ijt_files")

# Get the names of the extracted files
import glob
file_names = glob.glob("ijt_files/*.xml")

# Create the data set object
from openff.evaluator.datasets.thermoml import ThermoMLDataSet
data_set = ThermoMLDataSet.from_file(*file_names)

# Save the data set to a JSON object
data_set.json(file_path="ijt.json", format=True)
```

2.11 Taproom

The `TaproomDataSet` object offers an API for retrieving host-guest binding affinity measurements from the curated taproom repository.

Note: `taproom` may be installed by running `conda install -c conda-forge taproom`

This includes retrieving all of the data available:

```
from openff.evaluator.datasets.taproom import TaproomDataSet
taproom_set = TaproomDataSet()
```

data measure for a single host molecule (e.g. alpha-cyclodextrin):

```
acd_taproom_set = TaproomDataSet(host_codes=["acd"])
```

or data for a particular host and guest pair:

```
acd_taproom_set = TaproomDataSet(host_codes=["acd"], guest_codes=["bam"])
```

All measurements in this data set have an associated `TaproomSource` as their source provenance. This tracks both the original source of the measurement as well as the taproom identifier.

Note: Currently the data set object will assume a default set of buffer conditions (either no buffer, or a buffer of a salt with a specified ionic strength) rather than reading the buffer from the taproom measurement directory. This is consistent with previous applications of the data set.

2.12 Data Set Curation

The framework offers a full suite of features to facilitate the curation of data sets of physical properties, including:

- a significant amount of data filters, including to filter by state, substance composition and chemical functionalities.

and components to

- easily download and import the full *NIST ThermoML* and FreeSolv archives .
- select data points which were measured close to a set of target states, and which were measured for a diverse range of substances which contain specific functionalities.
- convert between different compatible property types (e.g. convert density <-> excess molar volume data).

These features are implemented as `CurationComponent` objects, which take as input an associated `CurationComponentSchema` which controls how the curation components should be applied to a particular data set (or a data set which is being stored as pandas `DataFrame` object).

An example of a curation component would be one that filters out data points which were measured outside of a particular temperature range:

```
# Filter data points measured at less than 290.0 K or greater than 320.0 K
filtered_frame = FilterByTemperature.apply(
    data_frame,
    FilterByTemperatureSchema(minimum_temperature=290.0, maximum_temperature=320.0),
)
```

Curation components can be conveniently chained together using a `CurationWorkflow` and an associated `CurationWorkflowSchema` so as to easily curated full training and testing data sets:

```
curation_schema = WorkflowSchema(
    component_schemas=[
        # Import the ThermoML archive.
        thermoml.ImportThermoMLDataSchema()
        # Filter out any measurements made for systems with more than two components
        filtering.FilterByNComponentsSchema(n_components=[1, 2]),
        # Remove any duplicate data.
        filtering.FilterDuplicatesSchema(),
        # Filter out data points measured away from ambient
        # and biologically relevant temperatures.
        filtering.FilterByTemperatureSchema(
            minimum_temperature=298.0, maximum_temperature=320.0
        ),
        # Retain only density and enthalpy of mixing data points.
        filtering.FilterByPropertyTypesSchema(
            property_types=["Density", "EnthalpyOfMixing"],
        ),
        # Select data points measured for alcohols, esters or mixtures of both.
        selection.SelectSubstancesSchema(
            target_environments=[
                ChemicalEnvironment.Alcohol,
                ChemicalEnvironment.CarboxylicAcidEster,
            ],
            n_per_environment=10,
```

(continues on next page)

(continued from previous page)

```
        ),
    ]
)

data_frame = Workflow.apply(pandas.DataFrame(), curation)
```

2.12.1 Examples

Data Extraction

- `ImportFreeSolv`: A component which will download the latest, full FreeSolv data set from the GitHub repository:

```
from openff.evaluator.datasets.curation.components.freesolv import (
    ImportFreeSolv,
    ImportFreeSolvSchema,
)

# Import the full FreeSolv data set.
data_frame = ImportFreeSolv.apply(pandas.DataFrame(), ImportFreeSolvSchema())
```

- `ImportThermoMLData`: A component which will download all *supported data* from the NIST ThermoML Archive:

```
from openff.evaluator.datasets.curation.components.thermoml import (
    ImportThermoMLData,
    ImportThermoMLDataSchema,
)

# Import all data collected from the IJT journal.
data_frame = ImportThermoMLData.apply(pandas.DataFrame(),␣
↪ImportThermoMLDataSchema())
```

Filtration

- `FilterDuplicates`: A component to remove duplicate data points (within a specified precision) from a data set:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterDuplicates,
    FilterDuplicatesSchema,
)

filtered_frame = FilterDuplicates.apply(data_frame, FilterDuplicatesSchema())
```

- `FilterByTemperature`: A component which will filter out data points which were measured outside of a specified temperature range:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByTemperature,
```

(continues on next page)

(continued from previous page)

```
    FilterByTemperatureSchema,
)

filtered_frame = FilterByTemperature.apply(
    data_frame,
    FilterByTemperatureSchema(minimum_temperature=290.0, maximum_temperature=320.0),
)
```

- `FilterByPressure`: A component which will filter out data points which were measured outside of a specified pressure range:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByPressure,
    FilterByPressureSchema,
)

filtered_frame = FilterByPressure.apply(
    data_frame,
    FilterByPressureSchema(minimum_pressure=100.0, maximum_pressure=140.0),
)
```

- `FilterByMoleFraction`: A component which will filter out data points which were measured outside of a specified mole fraction range:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByMoleFraction,
    FilterByMoleFractionSchema,
)

filtered_frame = FilterByMoleFraction.apply(
    data_frame, FilterByMoleFractionSchema(mole_fraction_ranges={2: [[(0.1, 0.3)]]})
)
```

- `FilterByRacemic`: A component which will filter out data points which were measured for racemic mixtures:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByRacemic,
    FilterByRacemicSchema,
)

filtered_frame = FilterByRacemic.apply(data_frame, FilterByRacemicSchema())
```

- `FilterByElements`: A component which will filter out data points which were measured for systems which contain specific elements:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByElements,
    FilterByElementsSchema,
)

filtered_frame = FilterByElements.apply(
    data_frame,
```

(continues on next page)

(continued from previous page)

```
    FilterByElementsSchema(allowed_elements=["C", "O", "H"]),
)
```

- `FilterByPropertyTypes`: A component which will apply a filter which only retains properties of specified types:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByPropertyTypes,
    FilterByPropertyTypesSchema,
)

# Retain only density measurements made for either pure or binary systems.
filtered_frame = FilterByPropertyTypes.apply(
    data_frame,
    FilterByPropertyTypesSchema(
        property_types=["Density"],
        n_components={"Density": [1, 2]},
    ),
)
```

- `FilterByStereochemistry`: A component which filters out data points measured for systems whereby the stereochemistry of a number of components is undefined:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByStereochemistry,
    FilterByStereochemistrySchema,
)

filtered_frame = FilterByStereochemistry.apply(
    data_frame, FilterByStereochemistrySchema()
)
```

- `FilterByCharged`: A component which filters out data points measured for substance where any of the constituent components have a net non-zero charge.:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByCharged,
    FilterByChargedSchema,
)

filtered_frame = FilterByCharged.apply(data_frame, FilterByChargedSchema())
```

- `FilterByIonicLiquid`: A component which filters out data points measured for substances which contain or are classed as an ionic liquids:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByIonicLiquid,
    FilterByIonicLiquidSchema,
)

filtered_frame = FilterByIonicLiquid.apply(data_frame, FilterByIonicLiquidSchema())
```

- `FilterBySmiles`: A component which filters the data set so that it only contains either a specific set of smiles, or does not contain any of a set of specifically excluded smiles:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterBySmiles,
    FilterBySmilesSchema,
)

filtered_frame = FilterBySmiles.apply(
    data_frame, FilterBySmilesSchema(smiles_to_include=["CCCO"]),
)
```

- `FilterBySmirks`: A component which filters a data set so that it only contains measurements made for molecules which contain (or don't) a set of chemical environments represented by SMIRKS patterns:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterBySmirks,
    FilterBySmirksSchema,
)

filtered_frame = FilterBySmirks.apply(
    data_frame, FilterBySmirksSchema(smirks_to_include=["[#6a]"]),
)
```

- `FilterByNComponents`: A component which filters out data points measured for systems with specified number of components:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByNComponents,
    FilterByNComponentsSchema,
)

filtered_frame = FilterByNComponents.apply(
    data_frame, FilterByNComponentsSchema(n_components=[1, 2])
)
```

- `FilterBySubstances`: A component which filters the data set so that it only contains properties measured for particular substances:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterBySubstances,
    FilterBySubstancesSchema,
)

filtered_frame = FilterBySubstances.apply(
    data_frame, FilterBySubstancesSchema(substances_to_include=[("CO", "C")])
)
```

- `FilterByEnvironments`: A component which filters a data set so that it only contains measurements made for substances which contain specific chemical environments:

```
from openff.evaluator.datasets.curation.components.filtering import (
    FilterByEnvironments,
    FilterByEnvironmentsSchema,
)
```

(continues on next page)

(continued from previous page)

```
filtered_frame = FilterByEnvironments.apply(
    data_frame,
    FilterByEnvironmentsSchema(
        environments=[
            ChemicalEnvironment.Aqueous,
            ChemicalEnvironment.Alcohol,
            ChemicalEnvironment.Amine,
        ]
    ),
)
```

Data Selection

- `SelectSubstances`: A component for selecting data points which were measured for specified number of maximally diverse systems containing a specified set of chemical functionalities:

```
# Select (if possible) data points which were measured for 10 different (and
# structurally diverse) alcohols.
schema = SelectSubstancesSchema(
    target_environments=[ChemicalEnvironment.Alcohol],
    n_per_environment=10,
)

data_frame = ConvertExcessDensityData.apply(data_frame, schema)
```

- `SelectDataPoints`: A component for selecting a set of data points which are close to a particular set of states:

```
# Select (if possible) density data points which were measured for pure systems
# at close to 298.15 K and 308.15K
schema = SelectDataPointsSchema(
    target_states=[
        TargetState(
            property_types=[("Density", 1)],
            states=[
                State(temperature=298.15, pressure=101.325, mole_fractions=(1.0,),
                State(temperature=308.15, pressure=101.325, mole_fractions=(1.0,),
            ],
        )
    ]
)

data_frame = ConvertExcessDensityData.apply(data_frame, schema)
```

Data Conversion

- `ConvertExcessDensityData`: A component for converting binary mass density data to excess molar volume data and vice versa where pure density data measured for the components is available:

```
from openff.evaluator.datasets.curation.components.conversion import (
    ConvertExcessDensityData,
    ConvertExcessDensityDataSchema,
)

converted_data_frame = ConvertExcessDensityData.apply(
    data_frame, ConvertExcessDensityDataSchema()
)
```

2.13 Physical Properties

A core philosophy of this framework is that users should be able to seamlessly curate data sets of physical properties and then estimate that data set using computational methods without significant user intervention and using sensible, well validated workflows.

This page aims to provide an overview of which physical properties are supported by the framework and how they are computed using the different *calculation layers*.

In this document $\langle X \rangle$ will be used to denote the ensemble average of an observable X.

2.13.1 Density

The density (ρ) is computed according to

$$\rho = \left\langle \frac{M}{V} \right\rangle$$

where M and V are the total molar mass and volume the system respectively.

Direct Simulation

The density is estimated using the default *simulation workflow* without modification. The estimation of liquid densities is assumed.

MBAR Reweighting

The density is estimated using the default *reweighting workflow* without modification. The estimation of liquid densities is assumed.

2.13.2 Dielectric Constant

The dielectric constant (ε) is computed from the fluctuations in a systems dipole moment (see Equation 7 of [1]) according to:

$$\varepsilon = 1 + \frac{\left\langle \vec{\mu}^2 \right\rangle - \left\langle \vec{\mu} \right\rangle^2}{3\varepsilon_0 \left\langle V \right\rangle k_b T}$$

where $\vec{\mu}$, V are the systems dipole moment and volume respectively, k_b the Boltzmann constant, T the temperature, and ε_0 the permittivity of free space.

Note: In *v0.2.2* and earlier of the framework the variance was computed as $\left\langle \left(\vec{\mu} - \left\langle \vec{\mu} \right\rangle\right)^2 \right\rangle$ in order to match the mdtraj implementation which has been used in previous studies by the OpenFF Consortium (see for example [2]). The two approaches should be numerically indistinguishable however.

Direct Simulation

The dielectric is estimated using the default *simulation workflow* which has been modified to use the specialized `AverageDielectricConstant` protocol in place of the default `AverageObservable` protocol. The estimation of liquid dielectric constants is assumed.

MBAR Reweighting

The dielectric is estimated using the default *reweighting workflow* which has been modified to use the specialized `ReweightDielectricConstant` protocol in place of the default `ReweightObservable` protocol. It should be noted that the `ReweightDielectricConstant` protocol employs bootstrapping to compute the uncertainty in the average dielectric constant, rather than attempting to propagate uncertainties in the average dipole moments and volumes. The estimation of liquid dielectric constants is assumed.

2.13.3 Enthalpy of Vaporization

The enthalpy of vaporization ΔH_{vap} (see [3]) can be computed according to

$$\Delta H_{vap} = \langle H_{gas} \rangle - \langle H_{liquid} \rangle = \langle E_{gas} \rangle - \langle E_{liquid} \rangle + p\left(\langle V_{gas} \rangle - \langle V_{liquid} \rangle\right)$$

where H, E, and V are the enthalpy, total energy and volume respectively.

Under the assumption that $V_{gas} >> V_{liquid}$ and that the gas is ideal the above expression can be simplified to

$$\Delta H_{vap} = \langle U_{gas} \rangle - \langle U_{liquid} \rangle + RT$$

where U is the potential energy, T the temperature and R the universal gas constant. This simplified expression is computed by default by this framework.

Direct Simulation

- **Liquid phase**: The potential energy of the liquid phase is estimated using the default *simulation workflow*, and divided by the number of molecules in the simulation box using the `divisor` input of the `AverageObservable` protocol.
- **Gas phase**: The potential energy of the gas phase is estimated using the default *simulation workflow*, which has been modified so that
  - the simulation box only contains a single molecule.
  - all periodic boundary conditions have been disabled.
  - all simulations are performed in the NVT ensemble.
  - the production simulation is run for 15000000 steps at a time (rather than 1000000 steps).
  - all simulations are run using the OpenMM reference platform (CPU only) regardless of whether a GPU is available. This is fastest platform to use when simulating a single molecule in vacuum with OpenMM.

The final enthalpy is then computed by subtracting the gas potential energy from the liquid potential energy (`SubtractValues`) and adding the RT term (`AddValues`). Uncertainties are propagated through the subtraction by the normal means using the uncertainties package.

MBAR Reweighting

- **Liquid phase**: The potential energy of the liquid phase is estimated using the default *reweighting workflow*, and divided by the number of molecules in the simulation box using an extra `DivideValue` protocol.
- **Gas phase**: The potential energy of the gas phase is estimated using the default *reweighting workflow*, which has been modified so that all periodic boundary conditions have been disabled.

The final enthalpy is then computed by subtracting the gas potential energy from the liquid potential energy (`SubtractValues`) and adding the RT term (`AddValues`). Uncertainties are propagated through the subtraction by the normal means using the uncertainties package.

2.13.4 Enthalpy of Mixing

The enthalpy of mixing $\Delta H_{mix}\left(x_0, \cdots, x_{M-1}\right)$ for a system of M components is computed according to

$$\Delta H_{mix}\left(x_0, \cdots, x_{M-1}\right) = \frac{\langle H_{mix} \rangle}{N_{mix}} - \sum_i^M x_i \frac{\langle H_i \rangle}{N_i}$$

where H_{mix} is the enthalpy of the full mixture, and H_i, x_i are the enthalpy and the mole fraction of component i respectively. N_{mix} and N_i are the total number of molecules used in the full mixture simulations and the simulations of each individual component respectively.

When re-weighting cached data to compute H_{mix} we make the approximation that the kinetic energy contributions cancel out between the mixture and each of the components, and hence can be computed by only re-weighting the NPT reduced potential:

$$\Delta H_{mix}\left(x_0, \cdots, x_{M-1}\right) \approx \frac{1}{\beta}\left(\frac{\langle u_{mix} \rangle}{N_{mix}} - \sum_i^M x_i \frac{\langle u_i \rangle}{N_i}\right)$$

where $u \equiv \beta\left(U + pV\right)$ is the NPT reduced potential, U the potential energy, p the pressure and V the volume.

Direct Simulation

- **Mixture**: The enthalpy of the full mixture is estimated using the default *simulation workflow* and divided by the number of molecules in the simulation box using the `divisor` input of the `AverageObservable` protocol.
- **Components**: The enthalpy of each of the components is estimated using the default *simulation workflow*, divided by the number of molecules in the simulation box using the `divisor` input of the `AverageObservable` protocol, and weighted by their mole fraction *in the mixture simulation box* using the `WeightByMoleFraction` protocol.

The final enthalpy is then computed by summing the component enthalpies (`AddValues`) and subtracting these from the mixture enthalpy (`SubtractValues`). Uncertainties are propagated through the summation and subtraction by the normal means using the uncertainties package.

MBAR Reweighting

- **Mixture**: The reduced potential of the full mixture is estimated using the default *reweighting workflow* and divided by the number of molecules in the reweighting box using an extra `DivideValue` protocol.
- **Components**: The reduced potential of each of the components is estimated using the default *reweighting workflow*, divided by the number of molecules in the reweighting box using an extra `DivideValue` protocol, and weighted by their mole fraction using the `WeightByMoleFraction` protocol.

The final enthalpy is then computed by summing the component enthalpies (`AddValues`), subtracting these from the mixture enthalpy (`SubtractValues`), and multiplying by $1/\beta$ (`MultiplyValue`). Uncertainties are propagated by the normal means using the uncertainties package.

2.13.5 Excess Molar Volume

The excess molar volume $\Delta V_{excess}\left(x_0, \cdots, x_{M-1}\right)$ for a system of M components is computed according to

$$\Delta V_{excess}\left(x_0, \cdots, x_{M-1}\right) = N_A \left(\frac{\langle V_{mix} \rangle}{N_{mix}} - \sum_i^M x_i \frac{\langle V_i \rangle}{N_i} \right)$$

where V_{mix} is the volume of the full mixture, and V_i, x_i are the volume and the mole fraction of component i respectively. N_{mix} and N_i are the total number of molecules used in the full mixture simulations and the simulations of each individual component respectively, and N_A is the Avogadro constant.

Direct Simulation

- **Mixture**: The molar volume of the full mixture is estimated using the default *simulation workflow* and divided by the molar number of molecules in the simulation box using the `divisor` input of the `AverageObservable` protocol.
- **Components**: The molar volume of each of the components is estimated using the default *simulation workflow*, divided by the molar number of molecules in the simulation box using the `divisor` input of the `AverageObservable` protocol, and weighted by their mole fraction *in the mixture simulation box* using the `WeightByMoleFraction` protocol.

The final excess molar volume is then computed by summing the component molar volumes (`AddValues`) and subtracting these from the mixture molar volume (`SubtractValues`). Uncertainties are propagated through the summation and subtraction by the normal means using the uncertainties package.

MBAR Reweighting

- **Mixture**: The enthalpy of the full mixture is estimated using the default *reweighting workflow* and divided by the molar number of molecules in the reweighting box using an extra `DivideValue` protocol.
- **Components**: The enthalpy of each of the components is estimated using the default *reweighting workflow*, divided by the molar number of molecules in the reweighting box using an extra `DivideValue` protocol, and weighted by their mole fraction using the `WeightByMoleFraction` protocol.

The final enthalpy is then computed by summing the component enthalpies (`AddValues`) and subtracting these from the mixture enthalpy (`SubtractValues`). Uncertainties are propagated through the summation and subtraction by the normal means using the uncertainties package.

2.13.6 Solvation Free Energies

Solvation free energies are currently computed using the Yank free energy package using direct molecular simulations. By default the calculations attempt to use 2000 solvent molecules, and the alchemical lambda spacings are selected using the built-in 'trailblazing' algorithm.

See the Yank documentation for more details.

2.13.7 Host-Guest Binding Free Energy

Warning: The computation of this property is still in beta. Users are heavily recommended to validate any calculations involving this property.

Host-guest binding free energies are currently computed using the attach-pull-release (APR) method [4] through integration with the pAPRika framework.

2.14 Common Workflows

As may be expected, most of the workflows used to estimate the physical properties within the framework make use of very similar workflows. This page aims to document the built-in 'template' workflows from which the more complex physical property estimation workflows are constructed.

2.14.1 Direct Simulation

Properties being estimated using the *direct simulation* calculation layer typically base their workflows off of the `generate_simulation_protocols()` template.

Note: This template currently assumes that a liquid phase property is being computed.

The workflow produced by this template proceeds as follows:

1) 1000 molecules are inserted into a simulation box with an approximate density of 0.95 g / mL using packmol (`BuildCoordinatesPackmol`).
2) the system is parameterized using either the *OpenFF toolkit*, *TLeap* or *LigParGen* depending on the force field being employed (`BuildSmirnoffSystem`, `BuildTLeapSystem` or `BuildLigParGenSystem`).

3) an energy minimization is performed using the default OpenMM energy minimizer (`OpenMMEnergyMinimisation`).

4) the system is equilibrated by running a short NPT simulation for 100000 steps using a timestep of 2 fs and using the OpenMM simulation engine (`OpenMMSimulation`).

5) while the uncertainty in the average observable is greater than the requested tolerance (if specified):

    5a) a longer NPT production simulation is run for 1000000 steps with a timestep of 2 fs and using the OpenMM simulation protocol (`OpenMMSimulation`) with its default Langevin integrator and Monte Carlo barostat.

    5b) the correlated samples are removed from the simulation outputs and the average value of the observable of interest and its uncertainty are computed by bootstrapping with replacement for 250 iterations (`AverageObservable`). See [1] for details of the decorrelation procedure.

    5c) steps 5a) and 5b) are repeated until the uncertainty condition (if applicable) is met.

The decorrelated simulation outputs are then made available ready to be cached by a *storage backend* (`DecorrelateObservables`, `DecorrelateTrajectory`).

2.14.2 MBAR Reweighting

Properties being estimated using the *MBAR reweighting* calculation layer typically base their workflows off of the `generate_reweighting_protocols()` template.

The workflow produced by this template proceeds as follows:

1) for each stored simulation data:

    1a) the cached data is retrieved from disk (`UnpackStoredSimulationData`)

2) the cached data from is concatenated together to form a single trajectory of configurations and observables (`ConcatenateTrajectories`, `ConcatenateStatistics`).

3) for each stored simulation data:

    3a) the system is parameterized using the force field parameters which were used when originally generating the cached data i.e. one of the reference states (`BuildSmirnoffSystem`, `BuildTLeapSystem` or `BuildLigParGenSystem`).

    3b) the reduced potential of each configuration in the concatenated trajectory is evaluated using the parameterized system (`OpenMMEvaluateEnergies`).

4) the system is parameterized using the force field parameters with which the property of interest should be calculated using i.e. of the target state (`BuildSmirnoffSystem`, `BuildTLeapSystem` or `BuildLigParGenSystem`) and the reduced potential of each configuration in the concatenated trajectory is evaluated using the parameterized system (`OpenMMEvaluateEnergies`).

    4a) *(optional)* if the observable of interest is a function of the force field parameters it is recomputed using the target state parameters. These recomputed values then replace the original concatenated observables loaded from the cached data.

5) the reference potentials, target potentials and the joined observables are sub-sampled to only retain equilibrated, uncorrelated samples (`AverageObservable`, `DecorrelateObservables`, `DecorrelateTrajectory`). See [1] for details of the decorrelation procedure.

6) the MBAR method is employed to compute the average value of the observable of interest and its uncertainty at the target state, taking the reference state reduced potentials as input. See [2] for the theory behind this approach. An exception is raised if there are not enough effective samples to reweight (`ReweightObservable`).

In more specialised cases the `generate_base_reweighting_protocols()` template (which `generate_reweighting_protocols()` is built off of) is instead used due to its greater flexibility.

2.14.3 References

2.15 Gradients

A most fundamental feature of this framework is its ability to rapidly compute the gradients of physical properties with respect to the force field parameters used to estimate them.

Note: Prior to v0.3.0 of this framework a combination of re-weighting and the central finite difference was employed to estimate the gradients of observables. From v0.3.0 onwards the fluctuation method [1] is instead used. The change was made to, in future, enable better integration with automatic differentiation libraries such as jax, and differentiable simulation engines such as timemachine.

2.15.1 Theory

The framework currently employs the fluctuation approach [1] to compute gradients of observables with respect to the force field parameters used to estimate them.

This approach may be derived by direct differentiation of the ensemble average an observable X:

$$\langle X\left(\theta\right)\rangle = \frac{1}{Q\left(\theta\right)} \int X\left(\theta\right) \exp\left[-\beta\left(U\left(\vec{r}, V; \theta\right) + pV\right)\right] \mathrm{d}\vec{r}\mathrm{d}V$$

where

$$Q\left(\theta\right) = \int \exp\left[-\beta\left(U\left(\vec{r}, V; \theta\right) + pV\right)\right] \mathrm{d}\vec{r}\mathrm{d}V$$

is the isothermal-isobaric partion function, θ are the force field parameters being used to estimate the observable, U the systems potential energy, $\beta \equiv k_b T$, k_b the Boltzmann constant, T the temperature, p the pressure and V the volume.

The derivative of the ensemble average defined above with respect to a particular force field parameter of interest θ is given by:

$$\frac{\mathrm{d}\left\langle X\right\rangle}{\mathrm{d}\theta_i} = \left\langle \frac{\mathrm{d}X}{\mathrm{d}\theta_i} \right\rangle - \beta \left[\left\langle X \frac{\mathrm{d}U}{\mathrm{d}\theta_i} \right\rangle - \left\langle \frac{\mathrm{d}U}{\mathrm{d}\theta_i} \right\rangle \langle X \rangle \right.$$

2.15.2 Computing $\mathrm{d}U/\mathrm{d}\theta_i$

While future integrations with differentiable simulation engines such as timemachine will allow $\mathrm{d}U/\mathrm{d}\theta_i$ to be computed directly from molecular simulation runs, currently most common simulation engines do not directly support computing this quantity.

Until such an integration is complete, the framework currently employs a central finite difference approach, whereby

$$\frac{\mathrm{d}U}{\mathrm{d}\theta_i} \approx \frac{U\left(\theta_i + h\right) - U\left(\theta_i - h\right)}{2h}$$

Although more expensive than computing either the forward or backwards derivative, the central difference method should give a more accurate estimate of the gradient at the minima, maxima and transition points. By default a value of $h = \theta_i \times 10^{-4}$ is used. This has been found to yield finite differences which do not suffer from precision issues, while being sufficiently small so as to yield an accurate estimate.

In practice the derivatives obtained by re-evaluating the energies of each configuration in a trajectory generated by a molecular simulation (either after a simulation or after loading one from disk) at each of the perturbed parameters.

While there is an expense associated with extra evaluations of the potential energy function for each configuration, this is mitigated by only computing those terms which depend upon (or may depend upon) θ_i. As an example, when computing derivatives with respect to a bond length the electrostatic and van der Waal contributions are not computed. This significantly speeds up the computation of these derivatives.

The final derivatives are stored in `ObservableArray` objects for convenience and for easy propagation of gradients through workflows. See the *observables documentation* for more information.

2.15.3 References

2.16 Calculation Layers

A `CalculationLayer` is an implementation of one calculation approach for estimating a set of physical properties, such as via molecular simulation or evaluating some QSAR like model.

The framework stacks multiple layers together when estimating a data set of properties.

Fig. 2: A schematic of the layer system. A set of properties to estimate are fed into the first layer. Those which can be calculated are returned back. Those that can't are passed to the next layer until no layer are left.

Each layer will in turn attempt to evaluate the properties being estimated using the specific approach the layer represents, such as by running a set of simulations. If the layer is unable to estimate a given property, for example if a layer does not yet support a given property, or if the layer has insufficient data to reprocesses, the property will be passed to the next layer for it to try and evaluate.

In practice, this allows the framework to attempt to estimate a data set using the most rapid calculation layer first, before moving to successively slower yet more robust layers, and thus enabling as efficient as possible property estimation.

2.16.1 Defining a Calculation Layer

A calculation layer is defined by two objects - a `CalculationLayer` object which implements the main layer logic, and a `CalculationLayerSchema` which defines those settings and options exposed required by the layer.

One `CalculationLayerSchema` will be provided to the for each type of property that the layer is being asked to estimate. The base `CalculationLayerSchema` currently only exposes options for optionally defining either the relative or absolute uncertainty that the layer should attempt to estimate the associated property type to within, however custom schemas can be defined per layer.

The structure of a `CalculationLayer` is relatively simple and permissive:

```
@calculation_layer()
class MyCalculationLayer(CalculationLayer):

    @classmethod
    def required_schema_type(cls):
        return CalculationLayerSchema

    @classmethod
    def _schedule_calculation(
        cls,
        calculation_backend,
        storage_backend,
        layer_directory,
        batch
    ):
        ...
```

The first thing to note is the `calculation_layer()` decorator which is being applied to the class. This registers the calculation layer with the frameworks plug-in system, allowing it to be used in future calculations.

The only other requirements is that the class implement a `required_schema_type` class method, which returns the type of `CalculationLayerSchema` that is associated with this layer, and a `_schedule_calculation()`. The `_schedule_calculation()` is responsible for performing the actual property calculations.

The form of the `_schedule_calculation()` function is very flexible:

```
@classmethod
def _schedule_calculation(
    cls,
    calculation_backend,
    storage_backend,
    layer_directory,
    batch
):

    futures = []

    for queued_property in batch.queued_properties:

        futures.append(
            calculation_backend.submit_task(
                cls.process_property, queued_property, cls.__name__
            )
```

(continues on next page)

(continued from previous page)

```
        )

    return futures
```

It takes as arguments:

- a *CalculationBackend* which is used to asynchronously distribute any calculations across the available compute resources.
- a *StorageBackend* which may be used to store / cache any data generated by the calculations.
- the path to the directory within which all of the calculation working files should be stored.
- the `Batch` of properties which this layer should attempt to estimate. This object includes the properties to estimate, as well as the `CalculationLayerSchema` for each property type.

and must return a list of `Future` objects (which either must be or implement the same API as the asyncio Future object). The easiest way to generate the futures is to perform any calculations using the `calculation_backend` which will automatically return the results of any functions as such.

The future objects returned by `_schedule_calculation()` must return a `CalculationLayerResult` object, which includes

- the estimated property if the calculation was successful (or `UNDEFINED` otherwise).
- a list of any exceptions (of type `EvaluatorException`) which were raised during the calculation.
- a list of any data to be stored by the storage backend.

As a minimal example of a method which returns one such object:

```
@classmethod
def process_property(cls, physical_property, **_):
    """Return a result as if the property had been successfully estimated.
    """

    # TODO: Do some calculations

    # Set the property provenance
    physical_property.source = CalculationSource(fidelity=cls.__name__)

    # Return the results object.
    results = CalculationLayerResult()
    results.physical_property = physical_property
    return results
```

2.16.2 Default Schemas

Default schemas for each pair of a calculation layer and a type of physical property may be registered using the `register_calculation_schema()` function:

```
# Register the default schema to use for density measurements being estimated
# by the direct simulation calculation layer.
register_calculation_schema(
    property_class=Density,
    layer_class=SimulationLayer,
```

(continues on next page)

(continued from previous page)

```
    schema=Density.default_simulation_schema
)
```

where the schema object should either be an instance of a `CalculationLayerSchema`, or a function with no required arguments which returns a `CalculationLayerSchema`.

A list of the registered schemas is provided by the `registered_calculation_schemas` module attribute.

2.17 Workflow Layers

The `WorkflowCalculationLayer` and `WorkflowCalculationSchema` offer an abstract base implementation for any calculation layers (and their associated schemas) which will perform their calculations using the built-in *workflow engine*.

The `WorkflowCalculationLayer` takes as input from its calculation schema one `WorkflowSchema` object for each type of property to be estimated by this layer. These schemas must *at a minimum* provide both the schemas of the protocols in the workflow, and have the `final_value_source` attribute set to the value of the calculated observable. In addition, the layer fully supports schemas which provide gradient information (see the `gradients_sources` attribute), as well as storing any generated dataclasses (see the `outputs_to_store` attribute) to the available storage backend.

This layer implements three key methods which are available to be overridden by any subclass implementations:

- `_get_workflow_metadata()`: a method which returns the dictionary of *metadata* which will be made available to the workflow (see the *default metadata* section for details).
- `_build_workflow_graph()`: the method which will construct the *workflow graph* to execute using the input workflow schemas and the metadata generated by the layer.
- `workflow_to_layer_result()`: a method which will map any `WorkflowResult` objects generated by the workflow graph into the `CalculationLayerResult` objects which the layer requires.

The workflow layer will by default tag each property estimated using it (or one of its derivatives) with a `CalculationSource` with the `fidelity` attribute set to the name of the layer, and the `provenance` attribute set to the schema of the workflow used to generate the property.

2.17.1 Default Metadata

The metadata provided to the workflows generated by this layer is generated on a per property to estimate basis mainly using the `generate_default_metadata()` function. It includes:

Key	Type	Description
`thermodynamic_state`	`ThermodynamicState`	The state at which the to perform any calculations .
`substance`	`Substance`	The substance to use in any calculations.
`components`	[`Substance`]	The components present in the main `substance`.
`target_uncertainty`	`Quantity`	The target uncertainty of any calculations defined by the calculation schema.
`per_component_uncertainty`	`Quantity`	The `target_uncertainty` divided by `sqrt(substance.n_components + 1)`
`force_field_path`	`str`	A file path to the force field parameters to use.
`parameter_gradient_keys`	[`ParameterGradientKey`]	The parameters to differentiate any observables with respect to (if any).

2.18 The Direct Simulation Layer

The `SimulationLayer` is a calculation layer which employs molecular simulation to estimate data sets of physical properties. It inherits the `WorkflowCalculationLayer` base layer, and primarily makes use of the built-in *workflow* engine to perform the required calculations.

The simulation layer is expected to *almost always* be able to estimate any properties requested of it (with exceptions being where a workflow schema has not yet been defined for a class of properties, or where an unexpected error occurs), and can be thought of as a safe 'fallback' layer when no other calculation approach are able to estimate particular properties.

It is expected that *workflow schemas* passed to the simulation layer should be able to estimate the gradients of the observable they aim to calculate, as well as specify a set of :doc:` storage/dataclasses <storage/dataclasses>` which contain the data generated by the molecular simulations.

2.18.1 Default Metadata

The simulation layer makes the same set of metadata available to its workflows as the *parent workflow layer*.

2.19 The MBAR Reweighting Layer

The `ReweightingLayer` is a calculation layer which employs the Multistate Bennett Acceptance Ratio (MBAR) method to calculate observables at states which have not been previously simulated, but for which simulations have been previously run at similar states and their data cached. It inherits the `WorkflowCalculationLayer` base layer, and primarily makes use of the built-in *workflow* engine to perform the required calculations.

Because MBAR is a technique which reprocesses exisiting simulation data rather than re-running new simulations, it is typically several fold faster than the *simulation layer* provided it has cached simulation data (made accessible via a *storage backend*) available. Any properties for which the required data (see *Calculation Schema*) is not available will be skipped.

2.19.1 Theory

The theory behind applying MBAR to reweighting observables from a simulated state to an unsimulated state is covered in detail in the publication Configuration-Sampling-Based Surrogate Models for Rapid Parameterization of Non-Bonded Interactions.

2.19.2 Calculation Schema

The reweighting layer will be provided with one `ReweightingSchema` per type of property that it is being requested to estimate. It builds off of the base `WorkflowCalculationSchema` schema providing an additional `storage_queries` attribute.

The `storage_queries` attribute will contain a dictionary of `SimulationDataQuery` which will be used by the layer to access the data required for each property from the storage backend. Each key in this dictionary will correspond to the key of a piece of metadata made available to the property workflows.

2.19.3 Default Metadata

The reweighting layer makes available the default metadata provided by the *parent workflow layer* in addition to any cached data retrieved via the schemas `storage_queries`.

When building the metadata for each property, a copy of the query will be made and any of the supported attributes (currently only `substance`) whose values are set as `PlaceholderValue` objects will have their values updated using values directly from the property. This query will then be passed to the storage backend to retrieve any matching data.

The matching data will be stored as a list of tuples of the form:

```
(object_path, data_directory, force_field_path)
```

where `object_path` is the file path to the stored dataclass, the `data_directory` is the file path to the ancillary data directory and `force_field_path` is the file path to the force field parameters which were used to generate the data originally.

This list of tuples will be made available as metadata under the key that was associated with the query.

2.20 Workflows

The framework offers a lightweight workflow engine for executing graphs of tasks using the available *calculation backends*. While lightweight, it offers a large amount of extensibility and flexibility, and is currently used by both the *simulation* and *reweighting* layers to perform their required calculations.

A workflow is a wrapper around a collection of tasks that should be executed in succession, and whose outputs should be made available as the input to others.

Fig. 3: A an example workflow which combines a protocol which will build a set of coordinates for a particular system, assign parameters to that system, and then perform an energy minimisation.

The workflow engine offers a number of advanced features such as the *automatic reduction of redundant tasks*, and *looping over parts of a workflow*

2.20.1 Building Workflows

At its core a workflow must define the tasks which need to be executed, and where the inputs to those tasks should be sourced from. Each task to be executed is represented by a *protocol object*, with each protocol requiring a specific set of user specified inputs:

```
# Define a protocol which will build some coordinates for a system.
build_coordinates = BuildCoordinatesPackmol("build_coordinates")
build_coordinates.max_molecules = 1000
build_coordinates.mass_density = 1.0 * unit.gram / unit.millilitre
build_coordinates.substance = Substance.from_components("O", "CO")

# Define a protocol which will assign force field parameters to the system.
assign_parameters = BuildSmirnoffSystem(f"assign_parameters")
assign_parameters.water_model = BuildSmirnoffSystem.WaterModel.TIP3P
assign_parameters.force_field_path = "openff-1.0.0.offxml"

# Set the `coordinate_file_path` input of the `assign_parameters` protocol
```

(continues on next page)

(continued from previous page)

```
# to the `coordinate_file_path` output of the `build_coordinates` protocol.
assign_parameters.coordinate_file_path = ProtocolPath(
    "coordinate_file_path", build_coordinates.id
)
```

The `ProtocolPath` object is used to reference the output of another protocol in the workflow, and will be replaced by the value of that output once that protocol has been executed by the workflow engine. It is constructed from two parts:

- the name of the output attribute to reference.
- the unique id of the protocol to take the output from.

To turn these tasks into a valid workflow which can be automatically executed, they must first be converted to a *workflow schema*:

```
# Create the schema object.
schema = WorkflowSchema()
# Add the individual protocol's schema representations to the workflow schema.
schema.protocol_schemas = [build_coordinates.schema, assign_parameters.schema]

# Create the executable workflow object from its schema.
workflow = Workflow.from_schema(schema, metadata=None)
```

A `Workflow` may either be synchronously executed in place yielding a `WorkflowResult` object directly:

```
workflow_result = workflow.execute()
```

or asynchronously using a calculation backend yielding a `Future` like object which will eventually return a `WorkflowResult`:

```
with DaskLocalCluster() as calculation_backend:
    result_future = workflow.execute(calculation_backend=calculation_backend)
```

In addition, a workflow may be add to, and executed as part as a larger *workflow graphs*.

2.20.2 Workflow Schemas

A `WorkflowSchema` is a blueprint from which all `Workflow` objects are constructed. It will predominantly define the tasks which compose the workflow, but may optionally define:

- `final_value_source`: A reference to the protocol output which corresponds to the value of the main observable calculated by the workflow.
- `gradients_sources`: A list of references to the protocol outputs which correspond to the gradients of the main observable with respect to a set of force field parameters.
- `outputs_to_store`: A list of *data classes* whose values will be populated from protocol outputs.
- `protocol_replicators`: A set of *replicators* which are used to flag parts of a workflow which should be replicated.

Each of these attributes will control whether the `value`, `gradients` and `data_to_store` attributes of the `WorkflowResult` results object will be populated respectively when executing a workflow.

Metadata

Because a schema is purely a blueprint for a general workflow, it need not define the exact values of all of the inputs of its constituent tasks. Consider the above example workflow for constructing a set of coordinates and assigning force field parameters to them. Ideally this one schema could be reused for multiple substances. This is made possible through a workflows *metadata*.

Each protocol within a workflow may access a dictionary of values unique to that workflow (termed here *metadata*) which is defined when the `Workflow` object is created from its schema.

This metadata may be accessed by protocols via a fictitious `"global"` protocol whose outputs map to the `metadata` dictionary:

```
build_coordinates = BuildCoordinatesPackmol("build_coordinates")
build_coordinates.substance = ProtocolPath("substance", "global")

# ...

substances = [
    Substance.from_components("CO"),
    Substance.from_components("CCO"),
    Substance.from_components("CCCO"),
]

for substance in substances:

    # Define the metadata to make available to the workflow protocols.
    metadata = {"substance": substance}
    # Create the executable workflow object from its schema.
    workflow = Workflow.from_schema(schema, metadata=metadata)

    # Execute the workflow ...
```

the created workflow will contain the `build_coordinates` protocol but with its `substance` input set to the value from the `metadata` dictionary.

2.21 Replicators

A `ProtocolReplicator` is the workflow equivalent of a `for` loop. It is statically evaluated when a `Workflow` is created from its schema. This is useful when parts of a workflow should be run multiple times but using different values for certain protocol inputs.

Note: The syntax of replicators is still rather rough around the edges, and will be refined in future versions of the framework.

Each `ProtocolReplicator` requires both a unique id and the set of *template values* which the replicator will 'loop' over to be defined. These values must either be a list of constant values or a reference to a list of values provided as *metadata*.

The 'loop variable' is referenced by protocols in the workflow using the `ReplicatorValue` placeholder input, where the value is linked to the replicator through the replicators unique id.

As an example, consider the case where a set of coordinates should be built for each component in a substance:

```
# Create the replicator object, and assign it a unique id.
replicator = ProtocolReplicator(replicator_id="component_replicator")
# Instruct the replicator to loop over all of the components of the substance
# made available by the global metadata
replicator.template_values = ProtocolPath("substance.components", "global")

# Define a protocol which will build some coordinates for a system.
build_coords = BuildCoordinatesPackmol("build_coords_" + replicator.placeholder_id})
# Instruct the protocol to use the value specified by the replicator.
build_coords.substance = ReplicatorValue(replicator.id)

# Build the schema containing the protocol and the replicator
schema = WorkflowSchema()
schema.protocol_schemas = [build_coords.schema]
schema.protocol_replicators = [replicator]
```

The requirement for a protocol to be replicated by a replicator is that its id *must* contain the replicators `placeholder_id` - this is a simple string which the workflow engine looks for when applying the replicator. The contents of this schema can be easily inspected by printing its JSON representation:

```
{
    "@type": "openff.evaluator.workflow.schemas.WorkflowSchema",
    "protocol_replicators": [
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolReplicator",
            "id": "component_replicator",
            "template_values": {
                "@type": "openff.evaluator.workflow.utils.ProtocolPath",
                "full_path": "global.substance.components"
            }
        }
    ],
    "protocol_schemas": [
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
            "id": "build_coords_$(component_replicator)",
            "inputs": {
                ".substance": {
                    "@type": "openff.evaluator.workflow.utils.ReplicatorValue",
                    "replicator_id": "component_replicator"
                }
            },
            "type": "BuildCoordinatesPackmol"
        }
    ]
}
```

It can be clearly seen that the schema only contains a single protocol entry, with the placeholder id present in its unique id. Once a workflow is created from this schema however:

```
# Define some metadata
metadata = {"substance": Substance.from_components("O", "CO")}
```

(continues on next page)

(continued from previous page)

```
# Build the workflow from the schema.
workflow = Workflow.from_schema(schema, metadata)
# Output the contents of the workflow as JSON.
print(workflow.schema.json())
```

it can be seen that the replicator has been correctly been applied and the workflow now contains one protocol for each component in the substance passed as metadata:


```
{
    "@type": "openff.evaluator.workflow.schemas.WorkflowSchema",
    "protocol_schemas": [
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
            "id": "build_coords_0",
            "inputs": {
                ".substance": {
                    "@type": "openff.evaluator.substances.components.Component",
                    "smiles": "O"
                }
            },
            "type": "BuildCoordinatesPackmol"
        },
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
            "id": "build_coords_1",
            "inputs": {
                ".substance": {
                    "@type": "openff.evaluator.substances.components.Component",
                    "smiles": "CO"
                }
            },
            "type": "BuildCoordinatesPackmol"
        }
    ]
}
```


In both cases the replicators `placeholder_id` has been replaced with the index of the value it was replicated for, and the substance input has been correctly set to the actual array value.

2.21.1 Nested Replicators

Replicators can be applied to other replicators to achieve a result similar to a set of nested for loops. For example the below loop:

```
components = [Component("O"), Component("CO")]
n_mols = [[1000], [500]]

for i, component in enumerate(components):

    for component_n_mols in n_mols[i]:

        ...
```

can readily be reproduced using replicators:

```
# Define a replicator which will loop over all components in the substance.
component_replicator = ProtocolReplicator(replicator_id="components")
component_replicator.template_values = ProtocolPath("components", "global")

# Define a replicator to loop over the number of each component to add.
n_mols_replicator_id = f"n_mols_{component_replicator.placeholder_id}"

n_mols_replicator = ProtocolReplicator(replicator_id=n_mols_replicator_id)
n_mols_replicator.template_values = ProtocolPath(
    f"n_mols[{component_replicator.placeholder_id}]", "global"
)

# Define the suffix which must be applied to protocols to be replicated
id_suffix = f"{component_replicator.placeholder_id}_{n_mols_replicator.placeholder_id}"

# Define a protocol which will build some coordinates for a system.
build_coordinates = BuildCoordinatesPackmol(f"build_coordinates_{id_suffix}")
build_coordinates.substance = ReplicatorValue(component_replicator.id)
build_coordinates.max_molecules = ReplicatorValue(n_mols_replicator.id)

# Build the schema containing the protocol and the replicator
schema = WorkflowSchema()
schema.protocol_schemas = [build_coordinates.schema]
schema.protocol_replicators = [component_replicator, n_mols_replicator]

# Define some metadata
metadata = {
    "components": [Component("O"), Component("CO")],
    "n_mols": [[1000], [500]]
}

# Build the workflow from the created schema.
workflow = Workflow.from_schema(schema, metadata)
# Print the JSON representation of the workflow.
print(workflow.schema.json(format=True))
```

Here the `component_replicator` placeholder id has been appended to the `n_mols_replicator` id to inform the workflow engine that the later is a child of the former. The `component_replicator` placeholder id is then used as an index into the `n_mols` array. This results in the following schema as desired:

```
{
    "@type": "openff.evaluator.workflow.schemas.WorkflowSchema",
    "protocol_schemas": [
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
            "id": "build_coordinates_0_0",
            "inputs": {
                ".max_molecules": 1000,
                ".substance": {
                    "@type": "openff.evaluator.substances.components.Component",
                    "smiles": "O"
                }
```

(continues on next page)

(continued from previous page)

```
            },
            "type": "BuildCoordinatesPackmol"
        },
        {
            "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
            "id": "build_coordinates_1_0",
            "inputs": {
                ".max_molecules": 500,
                ".substance": {
                    "@type": "openff.evaluator.substances.components.Component",
                    "smiles": "CO"
                }
            },
            "type": "BuildCoordinatesPackmol"
        }
    ]
}
```

2.22 Workflow Graphs

A `WorkflowGraph` is a collection of `Workflow` objects which should be executed together. The primary advantage of executing workflows via the graph object is that the graph will automatically take advantage of the *protocols* built in redundancy / merging support to collapse duplicate tasks across multiple workflows.

As an example, consider the case of executing workflows to estimate the density and the dielectric constant at the same state point, for the same substance, and using the same force field parameters:

```
density_schema = Density.default_simulation_schema()
dielectric_schema = DielectricConstant.default_simulation_schema()

density_workflow = Workflow.from_schema(density_schema, metadata)
dielectric_workflow = Workflow.from_schema(dielectric_schema, metadata)

print(len(density_workflow.protocols), len(dielectric_workflow.protocols))

workflow_graph = WorkflowGraph()
workflow_graph.add_workflows(density_workflow, dielectric_workflow)

print(len(workflow_graph.protocols))
```

The final workflow graph has roughly half the total number of density and dielectric protocols to be executed. This is expected as both the density and dielectric workflows are almost identical, except for the final analysis steps.

Graphs can be executed either in place without using a calculation backend in the same way that *workflows can*.

2.23 Protocols

The `Protocol` class represents a single task to be executed, whether that be as a standalone task or as a task which is part of some larger workflow. The task encoded by a protocol may be as simple as adding two numbers together or even as complex as performing entire free energy simulations:

```
from openff.evaluator.protocols.miscellaneous import AddValues

# Create the protocol and assign it some unique name.
add_numbers = AddValues(protocol_id="add_values")
# Set the numbers to add together
add_numbers.values = [1, 2, 3, 4]

# Execute the protocol
add_numbers.execute()

# Retrieve the output
result = add_numbers.result
```

2.23.1 Inputs and Outputs

Each protocol exposes a set of the required inputs as well as the produced outputs. These inputs may either be set as a constant directly, or if used as part of a *workflow*, can take their value from one of the outputs of another protocol.

Fig. 4: A selection of the inputs and outputs of the `OpenMMSimulation` protocol.

A surprisingly rich spectrum of workflows can be constructed by chaining together many relatively simple protocols.

The inputs and outputs of a protocol are defined using the custom `InputAttribute` and `OutputAttribute` descriptors:

```
class AddValues(Protocol):

    # Define the inputs that the protocol requires
    values = InputAttribute(
        docstring="The values to add together.",
        type_hint=list, default_value=UNDEFINED
    )

    # Define the outputs that the protocol will produce
    # once it is executed.
    result = OutputAttribute(
        docstring="The sum of the values.",
        type_hint=typing.Union[int, float, unit.Measurement, unit.Quantity],
    )

    def _execute(self, directory, available_resources):
        ...

    def validate(self, attribute_type=None):
        ...
```

Here we have defined a `values` input to the protocol and a `result` output. Both descriptors require a `docstring` and a `type_hint` to be provided.

The `type_hint` will be used by the workflow engine to ensure that a protocol which takes its input as the output of another protocol is receiving values of the correct type. Currently the `type_hint` can be any type of python class, or a `Union` of multiple types should the protocol allow for that.

In addition, the input attribute must specify a `default_value` for the attribute. This can either be a constant value, or a value set by some function such as a `lambda` statement:

```
some_input = InputAttribute(
    docstring="Takes it's default value from a function.",
    type_hint=int,
    default_value=lambda: return 1 + 1
)
```

In the above example we set the default value of `values` to `UNDEFINED` in order to specify that this input must be set by the user. The custom `UNDEFINED` class is used in place of `None` as `None` may be a valid input value for some attributes.

2.23.2 Task Execution

In addition to defining its inputs and outputs, a protocol must also implement an `_execute()` function which handles the main logic of the task:

```
def _execute(self, directory, available_resources):

    self.result = self.values[0]

    for value in self.values[1:]:
        self.result += value
```

The function is passed the directory in which it should run and create any working files, as well as a `ComputeResources` object which describes which compute resources are available to run on. This function *must* set all of the output attributes of the protocol before returning.

The private `_execute()` function which must be implemented should not be confused with the public `execute()` function. The public `execute()` function implements some common protocol logic (such as validating the inputs and creating the directory to run in if needed) before calling the private `_execute()` function.

2.23.3 Protocol Validation

The protocols inputs will automatically be validated before `_execute()` is called - this validation includes making sure that all of the non-optional inputs have been set, as well as ensuring they have been set to a value of the correct type. Protocols may implement additional validation logic by implementing a `execute()` function:

```
def validate(self, attribute_type=None):

    super(AddValues, self).validate(attribute_type)

    if len(self.values) < 1:
        raise ValueError("There were no values to add together")
```

2.23.4 Schemas

Every protocol has a `ProtocolSchema` representation which uniquely describes the protocol, and from which the protocol can be exactly recreated. The schema stores not only the type of protocol which it represents, but also the values of each of the inputs. Protocol schemas are fully JSON serializable. The following is an example schema for the above `add_numbers` protocol:

```
{
  "@type": "openff.evaluator.workflow.schemas.ProtocolSchema",
  "id": "add_values",
  "inputs": {
    ".allow_merging": true,
    ".values": [1, 2, 3, 4]
  },
  "type": "AddValues"
}
```

A protocols schema can be accessed via it's `schema` attribute. A protocol can be directly created from its schema representation by calling the schema's `to_protocol()` function.

2.23.5 Merging Protocols

When executing multiple workflows together (e.g. executing a workflow to estimate a substances density and potential energy) there is a large likelihood that some of tasks in those two workflows will be identical. Examples may include two workflows requiring protocols which build a set of coordinates, or assigning the same set of parameters to those coordinates.

Protocols have built-in support for comparing whether they are performing the same task / calculation as another protocol through the `can_merge()` and `merge()` functions:

- The `can_merge()` function checks to see whether two protocols are performing an identical task and hence whether they should be merged or not.
- The `merge()` function handles the actual merging of two protocols which can be merged.

The default `can_merge()` function takes advantage of the `merge_behvaiour` attribute of the different input descriptors. The `merge_behvaiour` attribute describes how each input should be considered when checking to see if two protocols can be merged:

```
max_molecules = InputAttribute(
    docstring="The maximum number of molecules to be added to the system.",
    type_hint=int,
    default_value=1000,
    merge_behavior=MergeBehaviour.ExactlyEqual
)
```

The most common behavior is to require that the inputs must be `ExactlyEqual` in order for two protocols two be considered to be identical. However, for some inputs such as the timestep of a simulation or the number of steps to simulate for, the exact values of the inputs don't necessarily need to be equal but rather, we may just wish to take the larger / smaller of the two inputs:

```
timestep = InputAttribute(
    docstring="The timestep to evolve the system by at each step.",
    type_hint=unit.Quantity,
    merge_behavior=InequalityMergeBehaviour.SmallestValue,
```

(continues on next page)

(continued from previous page)

```
    default_value=2.0 * unit.femtosecond,
)

total_number_of_iterations = InputAttribute(
    docstring="The number of times to propogate the system forward by.",
    type_hint=int,
    merge_behavior=InequalityMergeBehaviour.LargestValue,
    default_value=1,
)
```

This can be accomplished using the `InequalityMergeBehaviour` enum.

The default `merge()` function also relies upon the `merge_behaviour` attributes to determine which values of the inputs should be retained when merging two protocols.

2.24 Protocol Groups

The `ProtocolGroup` class represents a collection of *protocols* which have been grouped together. All protocols within a group will be executed together on a single compute resources, i.e. there is currently no support for executing protocols within a group in parallel.

Protocol groups have a specialised `ProtocolGroupSchema` which is essentially a collection of `ProtocolSchema` objects.

2.24.1 Conditional Protocol Groups

A `ConditionalGroup` is a special class of `ProtocolGroup` which will execute all of the grouped protocols again and again until a set of conditions has been met or until a maximum number of iterations (see `max_iterations`) has been performed. They can be thought of as being a protocol representation of a `while` statement.

Each condition to be met is represented by a `Condition` object:

```
condition = ConditionalGroup.Condition()

# Set the left and right hand values.
condition.left_hand_value = ...
condition.right_hand_value = ...

# Choose the type of condition
condition.type = ConditionalGroup.Condition.Type.LessThan
```

The left and right hand values can either be constants, or come from the output of another protocol (including grouped protocols) using a `ProtocolPath`. Currently a condition can either check that a value is less than or greater than another value.

Conditional groups expose a `current_iteration` attribute which tracks how many times the grouped protocols have been executed. This can be used as input by any of the grouped protocols and is useful, for example, to run a simulation for longer and longer until the groups condition has been met:

```
conditional_group = ConditionalGroup("conditional_group")

# Set up protocols to run a simulation and then to extract the
```

(continues on next page)

(continued from previous page)

```
# value of the density and its uncertainty.
simulation = OpenMMSimulation("simulation")
simulation.input_coordinate_file = "coords.pdb"
simulation.parameterized_system = ...

extract_density = AverageObservable("extract_density")
extract_density.observable = simulation.observables["Density"]

# Set the total number of iterations the simulation should perform to be equal
# to the current iteration of the group. I.e the simulation should perform a
# new iteration at each group iteration.
simulation.total_number_of_iterations = ProtocolPath(
    "current_iteration", conditional_group.id
)

# Add the protocols to the group.
conditional_group.add_protocols(production_simulation, analysis_protocol)

# Set up a condition which will check if the uncertainty is less than
# some threshold.
condition = ConditionalGroup.Condition()
condition.condition_type = groups.ConditionalGroup.Condition.Type.LessThan

condition.right_hand_value = 0.5 * unit.gram / unit.millilitre
condition.left_hand_value = ProtocolPath(
    "value.error", conditional_group.id, analysis_protocol.id
)

# Add the condition.
conditional_group.add_condition(condition)
```

It is this idea which is used to continue running a molecular simulations until an observable of interest (such as the density) has been calculated to within a specified uncertainty.

2.25 Observables

A key feature of this framework is its ability to compute the gradients of physical properties with respect to the force field parameters used to estimate them. This requires the framework be able to, internally, be able to not only track the gradients of all quantities which combine to yield the final observable of interest, but to also be able to propagate the gradients of those composite quantities through to the final value.

The framework offers three such objects to this end (`Observable`, `ObservableArray` and `ObservableFrame` objects) which will be covered in this document.

Note: In future versions of the framework the objects described here will likely be at least in part deprecated in favour of using full automatic differentiation libraries such as jax. Supporting these libraries will take a large re-write of the framework however, as well as full support between differentiable simulation engines like timemachine and the OpenFF toolkit. As such, these objects are implemented as stepping stones which can be gently phased out while working towards that larger, more modern goal.

2.25.1 Observable Objects

The base object used to track observables is the `Observable` object. It stores the average value, the standard error in the value and the gradient of the value with respect to force field parameters of interest.

Currently the value and error are internally stored in a composite `Measurement` object, which themselves wrap around the uncertainties package. This allows uncertainties to be automatically propagated through operations without the need for user intervention.

Note: Although uncertainties are automatically propagated, it is still up to property estimation workflow authors to ensure that such propagation (assuming a Gaussian error model) is appropriate. An alternative, which is employed throughout the framework is to make use of the bootstrapping technique.

Gradients are stored in a list as `ParameterGradient` gradient objects, which store both the floating value of the gradient alongside an identifying `ParameterGradientKey`.

Supported Operations

- **+ and -**: `Observable` objects can be summed with and subtracted from other `Observable` objects, `Quantity` objects, floats or integers. When two `Observable` objects are summed / subtracted, their gradients are combined by summing / subtracting also. When an `Observable` is summed / subtracted with a `Quantity`, `float` or `int` object it is assumed that these objects do not depend on any force field parameters.
- *: `Observable` objects may be multiplied by other `Observable` objects, `Quantity` objects, and `float` or `int` objects. When two `Observable` objects are multiplied their gradients are propagated using the product rule. When an `Observable` is multiplied by a `Quantity`, `float` or `int` object it is assumed that these objects do not depend on any force field parameters.
- /: `Observable` objects may be divided by other `Observable` objects, `Quantity` objects, and `float` or `int` objects. Gradients are propagated through the division using the quotient rule. When an `Observable` is divided by a `Quantity`, `float` or `int` object (or when these objects are divided by an `Observable` object) it is assumed that these objects do not depend on any force field parameters.

In all cases two `Observable` objects can only be operated on provided the contain gradient information with respect to the same set of force field parameters.

2.25.2 Observable Arrays

An extension of the `Observable` object is the `ObservableArray` object. Unlike an `Observable`, an `ObservableArray` object does not contain error information, but rather the value it stores and the gradients of that value should be a numpy array with `shape=(n_data_points, n_dimensions)`. It is designed to store information such as the potential energy evaluated at each configuration sampled during a simulation, as well as the gradient of the potential, which can then be ensemble averaged using a fluctuation formula to propagate the gradients through to the average.

Like with `Observable` objects, gradients are stored in a list as `ParameterGradient` gradient objects. The length of the gradients is required to match the length of the value array.

`ObservableArray` objects may be concatenated together using their `join()` method or sub-sampled using their `subset()` method.

Supported Operations

The `ObservableArray` object supports the same operations as the `Observable` object, whereby all operations are applied elementwise to the stored arrays.

2.25.3 Observable Frames

An `ObservableFrame` is a wrapper around a collection of `ObservableArray` which contain the types of observable specified by the `ObservableType` enum. It behaves as a dictionary which can take either an `ObservableType` or a string value of an `ObservableType` as an index.

Like an `ObservableArray`, observable frames may be concatenated together using their `join()` method or sub-sampled using their `subset()` method.

Supported Operations

No operations are supported between observable frames.

submit_task

2.26 Calculation Backends

A `CalculationBackend` is an object used to distribute calculation tasks across available compute resources. This is possible through specific backends which integrate with libraries such as multiprocessing, dask, parsl and cerlery.

Each backend is responsible for creating *compute workers*. A compute worker is an entity which has a set amount of dedicated compute resources available to it and which can execute python functions using those resources. Calculation backends may spawn multiple workers such that many tasks and calculations can be performed simultaneously.

A compute worker can be as simple as a new multiprocessing `Process` or something more complex like a dask worker. The resources available to a worker are described by the `ComputeResources` object.

`CalculationBackend` classes have a relatively simple structure:

```
class MyCalculationBackend(CalculationBackend):

    def __init__(self, number_of_workers, resources_per_worker):
        ...

    def start(self):
        ...

    def stop(self):
        ...

    def submit_task(self, function, *args, **kwargs):
        ...
```

By default they implement a constructor which takes as input the number of workers that the backend should initially spawn as well as the compute resources which are available to each. They must further implement:

- a `start()` method which spawns the initial set of compute workers.

- a `stop()` method which should kill all workers spawned by the backend as well as cleanup any temporary worker files.
- a `submit_task()` method which takes a function to be execute by a worker, and a set of `args` and `kwargs` to pass to that function.

The `submit_task()` must run asynchronously and return an asyncio `Future` object (or an object which implements the same API) when called, which can then be queried for when the task has completed.

All calculation backends are implemented as context managers such that they can be used as:

```
with MyCalculationBackend(number_of_workers=..., resources_per_worker...) as backend:
    backend.submit_task(...)
```

where the `start()` and `stop()` methods will be called automatically.

2.27 Dask Backends

The framework implements a number of calculation backends which integrate with the `dask` distributed and job-queue libraries.

2.27.1 Dask Local Cluster

The `DaskLocalCluster` backend wraps around the dask LocalCluster class to distribute tasks on a single machine:

```
worker_resources = ComputeResources(
    number_of_threads=1,
    number_of_gpus=1,
    preferred_gpu_toolkit=GPUToolkit.CUDA,
)

with DaskLocalCluster(number_of_workers=1, resources_per_worker=worker_resources) as␣
↪local_backend:
    local_backend.submit_task(logging.info, "Hello World")
    ...
```

Its main purpose is for use when debugging calculations locally, or when running calculations on machines with large numbers of CPUs or GPUs.

2.27.2 Dask HPC Cluster

The `DaskLSFBackend` and `DaskPBSBackend` backends wrap around the dask LSFCluster and PBSCluster classes respectively, and both inherit the `BaseDaskJobQueueBackend` class which implements the core of their functionality. They predominantly run in an adaptive mode, whereby the backend will automatically scale up or down the number of workers based on the current number of tasks that the backend is trying to execute.

These backends integrate with the queueing systems which most HPC cluster use to manage task execution. They work by submitting jobs into the queueing system which themselves spawn dask workers, which in turn then execute tasks on the available compute nodes:

```
# Create the object which describes the compute resources each worker should request from
# the queueing system.
```

(continues on next page)

(continued from previous page)

```
worker_resources = QueueWorkerResources(
    number_of_threads=1,
    number_of_gpus=1,
    preferred_gpu_toolkit=QueueWorkerResources.GPUToolkit.CUDA,
    per_thread_memory_limit=worker_memory,
    wallclock_time_limit="05:59",
)

# Create the backend object.
setup_script_commands = [
    f"conda activate evaluator",
    f"module load cuda/10.1",
]

calculation_backend = DaskLSFBackend(
    minimum_number_of_workers=1,
    maximum_number_of_workers=max_number_of_workers,
    resources_per_worker=queue_resources,
    queue_name="gpuqueue",
    setup_script_commands=setup_script_commands,
)

# Perform some tasks.
with calculation_backend:
    calculation_backend.submit_task(logging.info, "Hello World")
    ...
```

The `setup_script_commands` argument takes a list of commands which should be run by the queue job submission script before spawning the actual worker. This enables setting up custom environments, and setting any required environmental variables.

Configuration

To ensure optimal behaviour we recommend changing / uncommenting the following settings in the dask distributed configuration file (this can be found at `~/.config/dask/distributed.yaml`):

```
distributed:

    worker:
        daemon: False

    comm:
        timeouts:
            connect: 10s
            tcp: 30s

    deploy:
        lost-worker-timeout: 15s
```

See the dask documentation for more information about changing `dask` settings.

2.28 Storage Backends

A `StorageBackend` is an object used to store data generated as part of property calculations, and to retrieve that data for use in future calculations.

In general, most data stored in a storage backend is stored in two parts:

- A JSON serialized representation of this class (or a subclass), which contains lightweight information such as the state and composition of a system.
- A directory like structure (either directly a directory, or some NetCDF like compressed archive) of ancillary files which do not easily lend themselves to be serialized within a JSON object, such as simulation trajectories, whose files are referenced by their file name by the data object.

The ancillary directory-like structure is not required if the data may be suitably stored in the data object itself.

2.28.1 Data Storage / Retrieval

Each piece of data which is stored in a backend must inherit from the `BaseStoredData` class, will be assigned a unique key. This unique key is both useful for tracking provenance if this data is re-used in future calculations, and also can be used to retrieve the piece of data from the storage system.

In addition to retrieval using the data keys, each backend offers the ability to perform a 'query' to retrieve data which matches a set of given criteria. Data queries are implemented via `BaseDataQuery` objects, which expose different options for querying for specific types of data (such a simulation data, trained models, etc.).

A query may be used for example to match all simulation data that was generated for a given `Substance` in a particular phase:

```
# Look for all simulation data generated for liquid water
substance_query = SimulationDataQuery()

substance_query.substance = Substance.from_components("O")
substance_query.property_phase = PropertyPhase.Liquid

found_data = backend.query(substance_query)
```

The returned `found_data` will be a dictionary with keys of tuples and values as lists of tuples. Each key will be a tuple of the values which were matched, for example the matched thermodynamic state, or the matched substance. For each value tuple in the tuple list, the first item in the tuple is the unique key of the found data object, the second item is the data object itself, and the final object is the file path to the ancillary data directory (or `None` if none is present).

See the *Data Classes and Queries* page for more information about the available data classes, queries and their details.

2.28.2 Implementation

A `StorageBackend` must at minimum implement a structure of:

```
class MyStorageBackend(StorageBackend):

    def _store_object(self, object_to_store, storage_key=None, ancillary_data_path=None):
        ...

    def _retrieve_object(self, storage_key, expected_type=None):
        ...
```

(continues on next page)

(continued from previous page)

```
    def _object_exists(self, storage_key):
        ...
```

where

- `_store_object()` must store a `BaseStoredData` object as well as optionally its ancillary data directory, and return a unique key assigned to that object.
- `_retrieve_object()` must return the `BaseStoredData` object which has been assigned a given key if the object exists in the system, as well as the file path to ancillary data directory if it exists.
- `_object_exists()` should return whether any object still exists in the storage system with a given key.

All of these methods will be called under a reentrant thread lock and may be considered as thread safe.

2.29 Data Classes and Queries

All data which is to be stored within a `StorageBackend` must inherit from the `BaseStoredData` class. More broadly there are typically two types of data which are expected to be stored:

- `HashableStoredData` - data which is readily hashable and can be quickly queried for in a storage backend. The prime examples of such data are `ForceFieldData`, whose hash can be easily computed from the file representation of a force field.
- `ReplaceableData` - data which should be replaced in a storage backend when new data of the same type, but which has a higher information content, is stored in the backend. An example of this is when storing a piece of `StoredSimulationData` in the backend which was generated for a particular `Substance` and at the same `ThermodynamicState` as an existing piece of data, but which stores many more uncorrelated configurations.

Every data class **must** be paired with a corresponding data query class which inherits from the `BaseDataQuery` class. In addition, each data object must implement a `to_storage_query()` function which returns the data query which would uniquely match that data object. The `to_storage_query()` is used heavily by storage backends when checking if a piece of data already exists within the backend.

2.29.1 Force Field Data

The `ForceFieldData` class is used to `ForceFieldSource` objects within the storage backend. It is a hashable storage object which allows for rapidly checking whether any calculations have been previously been performed for a particular force field source.

It has a corresponding `ForceFieldQuery` class which can be used to query for particular force field sources within a storage backend.

2.29.2 Cached Simulation Data

Classes derived from the `BaseSimulationData` class are used to store the data generated by molecular simulation. The data object primarily records the `Substance`, `PropertyPhase` and `ThermodynamicState` that the simulation was run at, as well as provenance about the calculation and the force field parameters used (as the key of the force field in the storage system).

It has a corresponding `BaseSimulationDataQuery` class which can be used to query for simulation data which matches a set of particular criteria within a storage backend, which in part includes querying for data collected:

- at a given `thermodynamic_state` (i.e temperature and pressure).
- for a given `property_phase` (e.g. gas, liquid, liquid+gas coexisting, ...).
- using a given set of force field parameters identified by their unique `force_field_id` assigned by the storage system

Additionally included is not only the ability to find data generated for a particular `substance` (e.g. only data for methanol), but also the ability to return data for each component of a given substance by setting the `substance_query` attribute to a `SubstanceQuery` which has the `components_only` attribute set to true:

```
# Load an existing storage backend
storage_backend = LocalFileStorage()

# Define a system of 50% water and 50% methanol.
full_substance = Substance.from_components("O", "CO")

# Look for all simulation data generated for the full substance
data_query = SimulationDataQuery()

data_query.substance = full_substance
data_query.property_phase = PropertyPhase.Liquid

full_substance_data = storage_backend.query(data_query)

# Now look for all of the pure data which has been stored for both pure
# water and pure methanol.
pure_substance_query = SubstanceQuery()
pure_substance_query.components_only = True

data_query.substance_query = pure_substance_query
component_data = storage_backend.query(data_query)
```

This is particularly useful for when retrieving data for use in the calculation of excess properties (such as the enthalpy of mixing), where such calculations require information about both the full mixture as well as the pure components.

Single Simulation Data

The `StoredSimulationData` class is used to store data generated by a *single* molecular simulation and can be queried for using its accompanying `SimulationDataQuery` query class. In addition to the data stored by the parent `BaseSimulationData` class, this class further stores:

- the number of molecules which were simulated.
- the topology of the simulated system (stored as ancillary data).
- and trajectory of configurations (stored as ancillary data) and observables generated by the simulation.
- the statistic inefficiency of the data.

Data of this kind is considered replaceable, whereby data which has the lowest statistical efficiency is preferred. The philosophy here is that we should store the maximum amount of samples (i.e the maximum number of uncorrelated samples for the property which has the shortest correlation time) which will be useful for future calculations, such that future calaculations can simply discard the data which cannot be used (i.e. is likely correlated).

Free Energy Data

The `StoredFreeEnergyData` class is used to store data generated by a free energy calculation which computes the free energy difference between an end and start state. It can be queried for using its accompanying `FreeEnergyDataQuery` query class.

In addition to the data stored by the parent `BaseSimulationData` class, this class further stores:

- the free energy difference between the end and starting state.
- the topology of the system (stored as ancillary data).
- and trajectory of configurations generated in the starting and end states (stored as ancillary data).

Although data of this kind inherits from the `ReplaceableData` base class, all data deposited in a storage backend will be retained. At this time no situation can be envisaged that the same free energy data from exactly the same calculation will be stored, with the exception of operator errors.

2.30 Local File Storage

The `LocalFileStorage` backend stores and retrieves all data objects to / from the local file system. The root directory in which all data is to be stored is defined when the object is created:

```
storage_backend = LocalFileStorage(root_directory="stored_data")
```

All data objects will be stored within this directory as JSON files, with file names of the storage key assigned to that object. If the data object has an associated ancillary data directory, this will be **moved** (not copied) into the root directory and renamed to the storage key when that object is stored into the system.

An example directory created by a local storage backend will look something similar to:

```
- root_directory

    - 1fe615c5cb48429ab77fd71125dec297
        - trajectory.dcd
        - statistics.csv

    - 3e15d19e0e614d0491a1a0bc9a51534e
```

(continues on next page)

(continued from previous page)

```
      - trajectory.dcd
      - statistics.csv

  - 1fe615c5cb48429ab77fd71125dec297.json
  - 3e15d19e0e614d0491a1a0bc9a51534e.json
  - 0f71f2b4a22042d89d6f0882406869b6.json
```

where here the backend contains two data objects with ancillary data directories, and one without.

When retrieving data which has an ancillary data directory from the backend, the returned directory path will be the full path to the directory in the root storage directory.

2.31 Building the Docs

Although documentation for the OpenFF Evaluator is readily available online, it is sometimes useful to build a local version such as when

- developing new pages which you wish to preview without having to wait for ReadTheDocs to finish building.
- debugging errors which occur when building on ReadTheDocs.

In these cases, the docs can be built locally by doing the following:

```
git clone https://github.com/openforcefield/openff-evaluator.git
cd openff-evaluator/docs
conda env create --name openff-evaluator-docs --file environment.yaml
conda activate openff-evaluator-docs
rm -rf api && make clean && make html
```

The above will yield a new directory named *_build* which will contain the built html files which can be viewed in your local browser.

2.32 API

Documentation for each of the classes contained within the *openff.evaluator* framework.

2.32.1 Client Side API

Exceptions

2.32.2 Server Side API

2.32.3 Physical Property API

Built-in Properties

Substance Definition

State Definition

2.32.4 Data Set API

NIST ThermoML Archive

Taproom

Data Set Curation

Filtering

FreeSolv

ThermoML

Data Point Selection

Data Conversion

2.32.5 Force Field API

Gradient Estimation

2.32.6 Calculation Layers API

Built-in Calculation Layers

2.32.7 Calculation Backends API

Dask Backends

2.32.8 Storage API

Built-in Storage Backends

Data Classes

Data Queries

Attributes

2.32.9 Workflow API

Schemas

Attributes

Placeholder Values

2.32.10 Built-in Workflow Protocols

Analysis

Coordinate Generation

Force Field Assignment

Gradients

Groups

Miscellaneous

OpenMM

Paprika

Reweighting

Simulation

Storage

YANK Free Energies

2.32.11 Workflow Construction Utilities

2.32.12 Attribute Utilities

2.32.13 Observable Utilities

2.32.14 Plug-in Utilities

Plug-ins

2.33 Release History

Releases follow the `major.minor.micro` scheme recommended by PEP440, where

- `major` increments denote a change that may break API compatibility with previous `major` releases
- `minor` increments add features but do not break API compatibility
- `micro` increments represent bugfix releases or improvements in documentation

2.33.1 0.3.10

Bugfixes

- PR #444: Fix labelling molecules with virtual sites

2.33.2 0.3.9

Bugfixes

- PR #402: Fix importing full ThermoML archive

Behaviour Changes

The way that ThermoML archive files are served was changed in 2021 so that individual journal archives are no longer made available. Instead, now only the full ThermoML archive can be downloaded. Because of this, the `ImportThermoMLDataSchema` schema no longer allows users to select which journal to pull data from.

2.33.3 0.3.8

Bugfixes

- PR #390: Fix excluding v-sites from OpenMM positions

2.33.4 0.3.7

Bugfixes

- PR #389: Fix v-site positions not set by OpenMM

2.33.5 0.3.6

Bugfixes

- PR #375: Fix #374 - import from collections.abc
- PR #379: Fix #378 - 'FilterDuplicates` unintentionally selects values without uncertainty if multiple are present
- PR #384: Fix #382 - Default keyword arguments result in error
- PR #387: Fix #380 - Recursion error in local file storage

New Features

- PR #385: Support custom OpenMM nonbonded forces
- PR #386: Migrate to new OpenMM namespace

2.33.6 0.3.5

Bugfixes

- PR #367: Fix #365 - to/from_pandas does not roundtrip.
- PR #368: Fix #364 - Parsing an invalid IUPAC name raises an exception rather than a warning.
- PR #371: Fix gradients of non-Quantity parameters.

New Features

- PR #362: Support dask-jobqueue Slurm backend.
- PR #366: Support gradients of handler attributes.

2.33.7 0.3.4

A patch release which adds the option (and enables it by default) to remove working files, such as simulated trajectories, when they are no longer needed.

Behaviour Changes

- PR #349: Working files are deleted by default after an estimation batch completes.

2.33.8 0.3.3

This release facilitates the migration of the *openff-evaluator* package from *omnia* to *conda-forge*. This mainly involves changes which update the package to use the new namespaces introduced in the *openff-tookit* package, rather than the old and now deprecated *openforcefield* namespaces.

Bugfixes

- PR #346: Remove the unsupported *encoding* json kwarg.

New Features

- PR #341: Replace usages of dynamic Pint classes with internal static variants.
- PR #343: Migrate to the new OpenFF Toolkit namespace.
- PR #345: Migrate all reference from *omnia* to *conda-forge*.

2.33.9 0.3.2

This release exposes the option to disable caching of simulation data by an evaluator server. The performance of the local storage backend is currently poor when dealing with large amounts of cached data and hence it may be preferable to disable caching in such cases.

New Features

- PR #337: Expose server option to dis/enable data caching.

2.33.10 0.3.1

This release fixes a bug introduced in version 0.3.0 of this framework, whereby the default workflows for computing excess properties could in rare cases be incorrectly merged leading to downstream protocols taking their inputs from the wrong upstream protocol outputs.

While this bug should not affect most calculations, it is recommended that any production calculations performed using version 0.3.0 of this framework be repeated using version 0.3.1.

Bugfixes

- PR #331: Fixes merging excess properties.

2.33.11 0.3.0

The main feature of this release is the overhauling of how the framework computes the gradients of observables with respect to force field parameters.

In particular, from this release onwards all gradients will be computed using the fluctuation formula (also referred to as the thermodynamic gradient), rather than calculation be the re-weighted finite difference approach (PR #280). In general the two methods produce gradients which are numerically indistinguishable, and so this should not markedly change any scientific output of this framework.

The change was made to, in future, enable better integration with automatic differentiation libraries such as jax, and differentiable simulation engines such as timemachine which readily and rapidly give access to $\mathrm{d}U/\mathrm{d}\theta_i$.

Additionally, as of version 0.3.0 'known' charges (i.e. those assigned to TIP3P water and ions) are no longer automatically applied when using a SMIRNOFF based force field. This feature was originally included in the framework as the OpenFF toolkit did not support defining charges on specific molecules in the force field itself. This is now fully supported through the `LibraryCharges` section of a SMIRNOFF force field and hence this workaround is no longer required. From now on all ion and water charges **must** be specified in the SMIRNOFF force field.

Finally, this release includes **beta** support for computing host-guest binding affinities using the attach-pull-release (APR) method through integration with the pAPRika and taproom packages. This support was largely facilitated by the efforts of the `paprika` authors - David R. Slochower and Jeffry Setiadi.

Bugfixes

- PR #285: Use merged protocols in workflow provenance.
- PR #287: Fix merging of nested protocol inputs

New Features

- PR #262: Initial host-guest binding affinity support via `paprika` and `taproom`.
- PR #280: Switch to computing thermodynamic gradients.
- PR #309: Add a date to the timestamp logging output.
- PR #311: Initial solvation free energy gradient support.
- PR #312: Support caching free energy data.
- PR #324: Adds new miscellaneous `DummyProtocol` protocol.

Behaviour Changes

- PR #280: Migrate to thermodynamic gradients.
- PR #310: The SMIRNOFF protocol no longer applies 'known' charges (i.e. water and ions).
- PR #316: Add library charges to the TIP3P test data file.
- PR #328: Store workflow provenance as serialized string.

Breaking Changes

- The `StatisticsArray` array has been completely removed and replaced with a new set of observable (`Observable`, `ObservableArray`, `ObservableFrame` objects (#279, #286).
- The following protocol inputs / outputs have been renamed:
  - `SolvationYankProtocol.solvent_X_system` -> `SolvationYankProtocol.solution_X_system`
  - `SolvationYankProtocol.solvent_X_coordinates` -> `SolvationYankProtocol.solution_X_coordinates`
  - `SolvationYankProtocol.estimated_free_energy` -> `SolvationYankProtocol.free_energy_difference`
- The following classes have been renamed:
  - `OpenMMReducedPotentials` -> `OpenMMEvaluateEnergies`.
  - `AveragePropertyProtocol` -> `BaseAverageObservable`, `ExtractAverageStatistic` -> `AverageObservable`, `ExtractUncorrelatedData` -> `BaseDecorrelateProtocol`, `ExtractUncorrelatedTrajectoryData` -> `DecorrelateTrajectory`, `ExtractUncorrelatedStatisticsData` -> `DecorrelateObservables`
  - `ConcatenateStatistics` -> `ConcatenateObservables`, `BaseReducedPotentials` -> `BaseEvaluateEnergies`, `ReweightStatistics` -> `ReweightObservable`
- The following classes have been removed:
  - `OpenMMGradientPotentials`, `BaseGradientPotentials`, `CentralDifferenceGradient`

- The final value estimated by a workflow must now be an `Observable` object which contains any gradient information to return. (#296).

2.33.12 0.2.2

This release adds documentation for how physical properties are computed within the framework (both for this, and for previous releases.

Documentation

- PR #281: Initial pass at physical property documentation.

2.33.13 0.2.1

A patch release offering minor bug fixes and quality of life improvements.

Bugfixes

- PR #259: Adds `is_file_and_not_empty` and addresses OpenMM failure modes.
- PR #275: Workaround for N substance molecules > user specified maximum.

New Features

- PR #267: Adds workflow protocol to Boltzmann average free energies.
- PR #269: Expose exclude exact amount from max molecule cap.

2.33.14 0.2.0

This release overhauls the frameworks data curation abilities. In particular, it adds

- a significant amount of data filters, including to filter by state, substance composition and chemical functionalities.

and components to

- easily import all of the ThermoML and FreeSolv archives.
- convert between property types (currently density <-> excess molar volume).
- select data points close to a set of target states, and substances which contain specific functionalities (i.e. select only data points measured for ketones, alcohols or alkanes).

More information about the new curation abilities can be found *in the documentation here*.

New Features

- PR #260: Data set curation overhaul.
- PR #261: Adds `PhysicalPropertyDataSet.from_pandas`.

Breaking Changes

- All of the `PhysicalPropertyDataSet.filter_by_XXX` functions have now been removed in favor of the new curation components. See the *documentation* for information about the newly available filters and more.

2.33.15 0.1.2

A patch release offering minor bug fixes and quality of life improvements.

Bugfixes

- PR #254: Fix incompatible protocols being merged due to an id replacement bug.
- PR #255: Fix recursive `ThermodynamicState` string representation.
- PR #256: Fix incorrect version when installing from tarballs.

2.33.16 0.1.1

A patch release offering minor bug fixes and quality of life improvements.

Bugfixes

- PR #249: Fix replacing protocols of non-existent workflow schema.
- PR #253: Fix *antechamber* truncating charge file.

Documentation

- PR #252: Use *conda-forge* for *ambertools* installation.

2.33.17 0.1.0 - OpenFF Evaluator

Introducing the OpenFF Evaluator! The release marks a significant milestone in the development of this project, and constitutes an almost full redesign of the framework with a focus on stability and ease of use.

Note: *because of the extensive changes made throughout the entire framework, this release should almost be considered as an entirely new package. No files produced by previous versions of this will work with this new release.*

Clearer Branding

First and foremost, this release marks the complete rebranding from the previously named *propertyestimator* to the new *openff-evaluator* package. This change is accompanied by the introduction of a new `openff` namespace for the package, signifying it's position in the larger Open Force Field infrastructure and piplelines.

What was previously:

```
import propertyestimator
```

now becomes:

```
import openff.evaluator
```

The rebranded package is now shipped on `conda` under the new name of `openff-evaluator`:

```
conda install -c conda-forge -c omnia openff-evaluator
```

Markedly Improved Documentation

In addition, the release includes for the first time a significant amount of documentation for using the **`framework and it's features`_** as well as a collection of user focused tutorials which can be ran directly in the browser.

Support for RDKit

This release almost entirely removes the dependence on OpenEye thanks to support for RDKit almost universally across the framework.

The only remaining instance where OpenEye is still required is for host-guest binding affinity calculations where it is used to perform docking.

Model Validation

Starting with this release almost all models, range from `PhysicalProperty` entries to `ProtocolSchema` objects, are now heavily validated to help catch any typos or errors early on.

Batching of Similar Properties

The `EvaluatorServer` now more intelligently attempts to batch properties which may be computed using the same simulations into a single batch to be estimated. While the behaviour was already supported for pure properties in previous, this has now been significantly expanded to work well with mixture properties.

2.33.18 0.0.9 - Multi-state Reweighting Fix

This release implements a fix for calculating the gradients of properties being estimated by reweighting data cached from multiple independant simulations.

Bugfixes

- PR #143: Fix for multi-state gradient calculations.

2.33.19 0.0.8 - ThermoML Improvements

This release is centered around cleaning up the ThermoML data set utilities. The main change is that ThermoML archive files can now be loaded even if they don't contain measurement uncertainties.

New Features

- PR #142: ThermoML archives without uncertainties can now be loaded.

Breaking Changes

- PR #142: All *ThermoMLXXX* classes other than *ThermoMLDataSet* are now private.

2.33.20 0.0.7 - Bug Quick Fixes

This release aims to fix a number of minor bugs.

Bugfixes

- PR #136: Fix for comparing thermodynamic states with unset pressures.
- PR #138: Fix for a typo in the maximum number of minimization iterations.

2.33.21 0.0.6 - Solvation Free Energies

This release centers around two key changes -

i) a general refactoring of the protocol classes to be much cleaner and extensible through the removal of the old stub functions and the addition of cleaner descriptors.

ii) the addition of workflows to estimate solvation free energies via the new `SolvationYankProtocol` and `SolvationFreeEnergy` classes.

The implemented free energy workflow is still rather basic, and does not yet support calculating parameter gradients or estimation from cached simulation data through reweighting.

A new table has been added to the documentation to make clear which built-in properties support which features.

New Features

- PR #110: Cleanup and refactor of protocol classes.
- PR #125: Support for PBS based HPC clusters.
- PR #127: Adds a basic workflow for estimating solvation free energies with YANK.
- PR #130: Adds a cleaner mechanism for restarting simulations from checkpoints.
- PR #134: Update to a more stable dask version.

Bugfixes

- PR #128: Removed the defunct dask backend *processes* kwarg.
- PR #133: Fix for tests failing on MacOS due to *travis* issues.

Breaking Changes

- PR #130: The `RunOpenMMSimulation.steps` input has now been split into the `steps_per_iteration` and `total_number_of_iterations` inputs.

Migration Guide

This release contained several public API breaking changes. For the most part, these can be remedied by the follow steps:

- Replace all instances of `run_openmm_simulation_protocol.steps` to `run_openmm_simulation_protocol.steps_per_iteration`

2.33.22 0.0.5 - Fix For Merging of Estimation Requests

This release implements a fix for a major bug which caused incorrect results to be returned when submitting multiple estimation requests at the same time - namely, the returned results became jumbled between the different requests. As an example, if a request was made to estimate a data set using the *smirnoff99frosst* force field, and then straight after with the *gaff 1.81* force field, the results of the *smirnoff99frosst* request may contain some properties estimated with *gaff 1.81* and vice versa.

This issue does not affect cases where only a single request was made and completed at a time (i.e the results of the previous request completed before the next estimation request was made).

Bugfixes

- PR #119: Fixes gather task merging.
- PR #121: Update to distributed 2.5.1.

2.33.23 0.0.4 - Initial Support for Non-SMIRNOFF FFs

This release adds initial support for estimating property data sets using force fields not based on the `SMIRNOFF` specification. In particular, initial AMBER force field support has been added, along with a protocol which applies said force fields using `tleap`.

New Features

- PR #96: Adds a mechanism for specifying force fields not in the `SMIRNOFF` spec.
- PR #99: Adds support for applying `AMBER` force field parameters through `tleap`
- PR #111: Protocols now stream trajectories from disk, rather than pre-load the whole thing.
- PR #112: Specific types of protocols can now be easily be replaced using `WorkflowOptions`.
- PR #117: Adds support for converting `PhysicalPropertyDataSet` objects to `pandas.DataFrame`.

Bugfixes

- PR #115: Fixes caching data for substances whose smiles contain forward slashes.
- PR #116: Fixes inconsistent mole fraction rounding.

Breaking Changes

- PR #96: The `PropertyEstimatorClient.request_estimate(force_field=...` argument has been renamed to `force_field_source`.

Migration Guide

This release contained several public API breaking changes. For the most part, these can be remedied by the follow steps:

- Change all instances of `PropertyEstimatorClient.request_estimate(force_field=...)` to `PropertyEstimatorClient.request_estimate(force_field_source=...)`

2.33.24 0.0.3 - ExcessMolarVolume and Typing Improvements

This release implements a number of bug fixes and adds two key new features, namely built in support for estimating excess molar volume measurements, and improved type checking for protocol inputs and outputs.

New Features

- PR #98: `Substance` objects may now have components with multiple amount types.
- PR #101: Added support for estimating `ExcessMolarVolume` measurements from simulations.
- PR #104: `typing.Union` is now a valid type arguemt to `protocol_output` and `protocol_input`.

Bugfixes

- PR #94: Fixes exception when testing equality of `ProtocolPath` objects.
- PR #100: Fixes precision issues when ensuring mole fractions are <= *1.0*.
- PR #102: Fixes replicated input for children of replicated protocols.
- PR #105: Fixes excess properties weighting by the wrong mole fractions.
- PR #107: Fixes excess properties being converged to the wrong uncertainty.
- PR #108: Fixes calculating MBAR gradients of reweighted properties.

Breaking Changes

- PR #98: `Substance.get_amount` renamed to `Substance.get_amounts` and now returns an immutable `frozenset` of `Amount` objects, rather than a single `Amount`.
- PR #104: The `DivideGradientByScalar`, `MultiplyGradientByScalar`, `AddGradients`, `SubtractGradients` and `WeightGradientByMoleFraction` protocols have been removed. The `WeightQuantityByMoleFraction` protocol has been renamed to `WeightByMoleFraction`.

Migration Guide

This release contained several public API breaking changes. For the most part, these can be remedied by the follow steps:

- Change all instances of `Substance.get_amount` to `Substance.get_amounts` and handle the newly returned frozenset of amounts, rather than the previously returned single amount.
- Replace the now removed protocols as follows:
  - `DivideGradientByScalar` -> `DivideValue`
  - `MultiplyGradientByScalar` -> `MultiplyValue`
  - `AddGradients` -> `AddValues`
  - `SubtractGradients` -> `SubtractValues`
  - `WeightGradientByMoleFraction` -> `WeightByMoleFraction`
  - `WeightQuantityByMoleFraction` -> `WeightByMoleFraction`

2.33.25 0.0.2 - Replicator Quick Fixes

A minor release to fix a number of minor bugs related to replicating protocols.

Bugfixes

- PR #90: Fixes merging gradient protocols with the same id.
- PR #92: Fixes replicating protocols for more than 10 template values.
- PR #93: Fixes `ConditionalGroup` objects losing their conditions input.

2.33.26 0.0.1 - Initial Release

The initial pre-alpha release of the framework.

2.34 Release Process

This document aims to outline the steps needed to release the `openff-evaluator` on `conda-forge`. This should only be done with the approval of the core maintainers.

2.34.1 1. Update the Release History

If no PR has been submitted, create a new one to keep track of changes to the release notes *only*. Only the `releasehistory.rst` file may be edited in this PR.

Ensure that the release history file is up to date, and conforms to the below template:

```
X.Y.Z - Descriptive Title
-------------------------------

This release...

New Features
""""""""""""

* PR #X: Feature summary

Bugfixes
""""""""

* PR #Y: Fix Summary

Breaking Changes
""""""""""""""""

* PR #Z: Descriptive summary of the breaking change

Migration Guide
"""""""""""""""

This release contained several public API breaking changes. For the most part, these can␣
↪be
remedied by the follow steps:

* A somewhat verbose guide on how users should upgrade their code given the new breaking␣
↪changes.
```

(continues on next page)

(continued from previous page)

2.34.2 2: Cut the Release on GitHub

To cut a new release on GitHub:

1) Go to the `Releases` tab on the front page of the repo and choose `Create a new release`.

2) Set the release tag using the form: `X.Y.Z`

3) Added a descriptive title using the form: `X.Y.Z [Descriptive Title]`

4) Ensure the `This is a pre-release` checkbox is ticked.

5) Reformat the release notes from part 1) into markdown and paste into the description box.

a) Append the following extra message above the *New Features* title:

```
A richer version of these release notes with live links to API documentation is available
on [our ReadTheDocs page](https://property-estimator.readthedocs.io/en/latest/
↪releasehistory.html)

See our [installation instructions](https://property-estimator.readthedocs.io/en/latest/
↪install.html).

Please report bugs, request features, or ask questions through our
[issue tracker](https://github.com/openforcefield/openff-evaluator/issues).

**Please note that this is a pre-alpha release and there will still be major changes to␣
↪the API
prior to a stable 1.0.0 release.**
```

Note - You do not need to upload any files. The source code will automatically be added as a `.tar.gz` file.

2.34.3 3: Trigger a New Build on Conda Forge

To trigger the build on `conda-forge`:

1) Create a fork of the openff-evaluator-feedstock and make the following changes to the `recipe/meta.yaml` file:

a) Update the `version` to match the release.

b) Set `build` to 0

c) Update any dependencies in the `requirements` section

d) Update the sha256 hash to the output of `curl -sL https://github.com/openforcefield/openff-evaluator/archive/{{ version }}.tar.gz | openssl sha256`

2) Open PR to merge the fork into the main feedstock:

a) The PR title should have the format `Release X.Y.Z`

b) No PR body text is needed

c) The CI will run on this PR (~30 minutes) and attempt to build the package.

d) If the build is successful the PR should be reviewed and merged by the feedstock maintainers.

e) **Once merged** the package is built again on and uploaded to anaconda.

3) Test the `conda-forge` package:

a) `conda install -c conda-forge openff-evaluator`

2.34.4 4: Update the ReadTheDocs Build Versions

To ensure that the read the docs pages are updated:

1) Trigger a RTD build of `latest`.
2) Under the `Versions` tab add the new release version to the list of built versions and **save**.
3) Verify the new version docs have been built and pushed correctly
4) Under `Admin` | `Advanced Settings`: Set the new release version as Default version to display and **save**.

BIBLIOGRAPHY

[1] Alice Glättli, Xavier Daura, and Wilfred F van Gunsteren. Derivation of an improved simple point charge model for liquid water: spc/a and spc/l. *The Journal of chemical physics*, 116(22):9811–9828, 2002.

[2] Kyle A Beauchamp, Julie M Behr, Ariën S Rustenburg, Christopher I Bayly, Kenneth Kroenlein, and John D Chodera. Toward automated benchmarking of atomistic force fields: neat liquid densities and static dielectric constants from the thermoml data archive. *The Journal of Physical Chemistry B*, 119(40):12912–12920, 2015.

[3] Junmei Wang and Tingjun Hou. Application of molecular dynamics simulations in molecular property prediction. 1. density and heat of vaporization. *Journal of chemical theory and computation*, 7(7):2151–2165, 2011.

[4] David R Slochower, Niel M Henriksen, Lee-Ping Wang, John D Chodera, David L Mobley, and Michael K Gilson. Binding thermodynamics of host–guest systems with smirnoff99frosst 1.0. 5 from the open force field initiative. *Journal of Chemical Theory and Computation*, 15(11):6225–6242, 2019.

[1] John D Chodera. A simple method for automated equilibration detection in molecular simulations. *Journal of chemical theory and computation*, 12(4):1799–1805, 2016.

[2] Richard A Messerly, S Mostafa Razavi, and Michael R Shirts. Configuration-sampling-based surrogate models for rapid parameterization of non-bonded interactions. *Journal of Chemical Theory and Computation*, 14(6):3144–3162, 2018.

[1] Lee-Ping Wang, Teresa Head-Gordon, Jay W Ponder, Pengyu Ren, John D Chodera, Peter K Eastman, Todd J Martinez, and Vijay S Pande. Systematic improvement of a classical molecular model of water. *The Journal of Physical Chemistry B*, 117(34):9956–9972, 2013.