SI: *PySurf* - A Framework for Database Accelerated Direct Dynamics

Maximilian F.S.J. Menger,* Johannes Ehrmaier, and Shirin Faraji*

Zernike Institute for Advanced Materials, Faculty of Science and Engineering, University of Groningen, Nijenborgh 4, 9747AG Groningen, The Netherlands.

E-mail: m.f.s.j.menger@rug.nl; s.s.faraji@rug.nl

Contents

1 SO_2 calculations **2**

1.1 SO_2 LVC calculations 3

2 Plugin **4**

2.1 Tutorial: Add a Model System 4

2.2 Tutorial: Write an Ab-Initio Interface 9

2.3 Tutorial: Write an Interpolator 15

3 Workflow **22**

3.1 Tutorial: How to use the Workflow engine 22

3.2 Tutorial: Write your own Workflow 23

1 SO_2 calculations

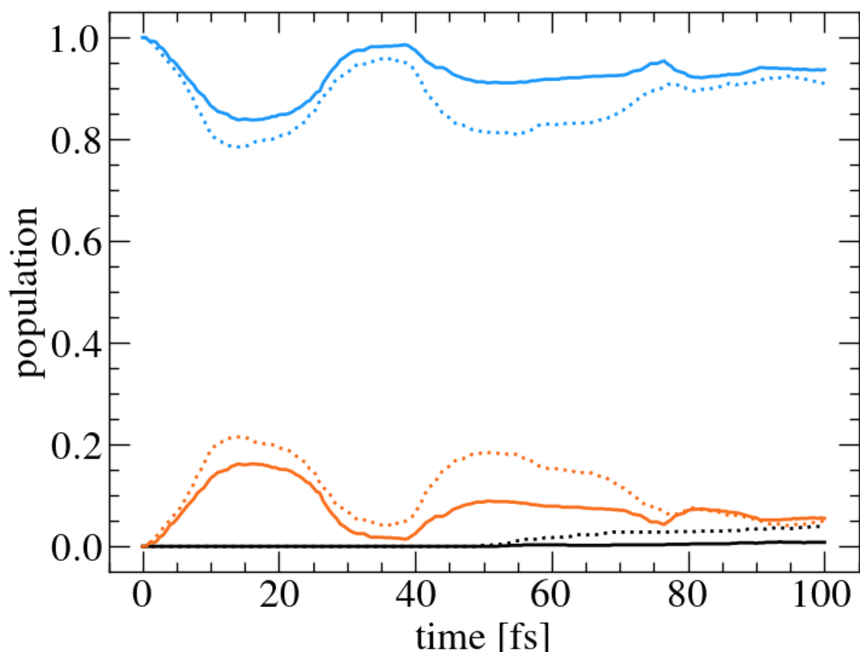

Figure1: Populations for SO_2 starting from the S_1 state for the full QM (solid) and *energy-only* (dotted) simulations with 1000 trajectories. black: S_0; blue: S_1; orange: S_2;

Fig. 1 shows the populations of SO_2 for 1000 reference calculations (solid) compared to *energy-only* simulations (dotted). Population transfer is observed from the S_1 to the S_2 state within the first 20 fs. After 30 fs most of the population is back in the S_1 state. For the *energy-only* simulations the population transfer is more extended. After 100 fs more than 90% of the population is in the S_1 state, which is similar to the reference calculations. The results are consistent with previous investigations based on wavepacket propagation or trajectory-surface hopping simulations.[1–3] The main differences can be explained by the different electronic-structure methods being used.

1.1 SO_2 LVC calculations

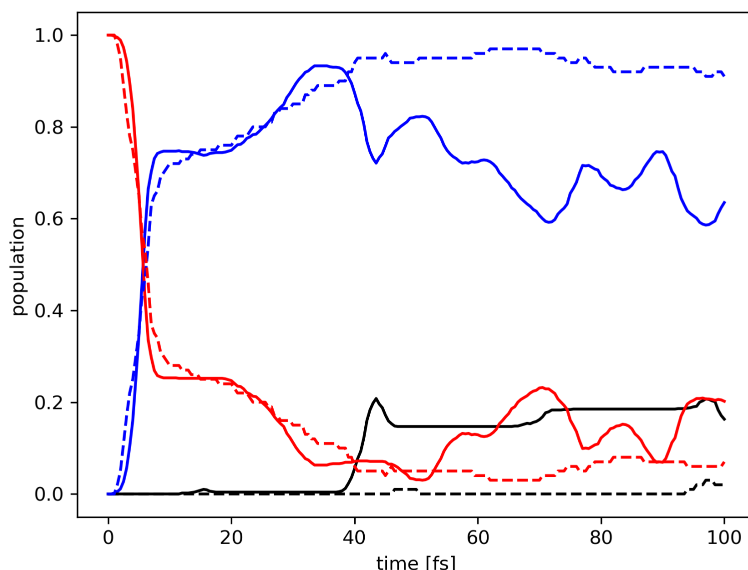

Figure2: Populations for SO_2 starting from the S_2 state for the linear vibronic coupling (solid) and the full QM (dashed) using Landau-Zener for simulations with 1000 trajectories. black: S_0; blue: S_1; red: S_2;

Fig. 2 shows the populations of SO_2 for linear vibronic coupling (LVC) calculations (solid) using the parameters presented in Ref. 4 (obtained at MR-CIS(6,6)/VDZP level of theory) and considering only the singlet excited states are compared to the full QM (TDDFT/B3LYP) simulations (dashed). In both cases the Landau-Zener surface hopping algorithm is used. and 1000 trajectories run. The population transfer is observed from the S_2 to the S_1 state within the first 20 fs. for the first 40 fs both simulations give almost the same results and deviations start to appear for longer simulation times. The results are consistent with previous investigations based on wavepacket propagation or trajectory-surface hopping simulations.[1–4] The main differences can be explained by the different electronic-structure methods being used.

2 Plugin

Extension of PySurf is mainly done via plugins. It is straightforward to add your own custom plugins to the PySurf framework. In the following we walk you through the process of creating a few example plugins to illustrate the task.

2.1 Tutorial: Add a Model System

Models in PySurf have the dimensionality (N,1). For example, Some pyrazine models in normal mode coordinates are already available in the package. To write your own model you simply have to create a child-class of the PySurf *Model* class and put your python script into the plugin folder, so that it gets automatically imported on start and is available in the whole environment.

2.1.1 Example: 1D Harmonic Oscillator

In this tutorial, we will implement a simple model with one or optionally two harmonic oscillators. The user can decide, whether the second surface is included and how it should be shifted with respect to the first surface. Additionally, the frequencies and the offset energy have to be specified from the user.

In the following we will walk you through the complete implementation. First we will import some helper classes from Pysurf, namely the *Model* class, which all models inherit from. Additionally we use the Mode class, which provides some utilities to deal with normal modes.

```
from pysurf import Model
from pysurf.system import Mode
```

Basic Features

The **Model** is an abstract base class and all children of it have to provide the following class attributes:

implemented , list of strings, defines which properties are provided by the model, e.g.

['energy', 'gradient', 'fosc']

modes list of the ground state modes.

Additionally, two class methods need to be added: 'from_config': which acts as a static constructor to instantiate the class using the user data.

from_config: which acts as a static constructor to instantiate the class using the user data.

get: fill the request with the demanded data

```
class HarmonicOscillator1D():
    """ Model for a 1D harmonic oscillator with 1 or 2 PES """

    implemented = ['energy', 'gradient']
    crd = [0.0]
    frequencies = [1.0]
    displacements = [[1.0]]
    modes = [Mode(freq, dis) for freq, dis in zip(frequencies, displacements)]

    @classmethod
    def from_config(cls, config):
        """initialize class with a given questions config!"""

    def get(self, request):
        """get requested info"""
```

User Input

PySurf is build around the Colt framework, developed independently. To specify additional user input for your class you simply define a *_questions* string as a class attribute.

In our example we will need user input for the shape of the potentials:

- **E0/E1** offset energy of the corresponding potential

- **x0/x1** shift of the harmonic oscillator in x direction
- **w0/w1** frequency of the corresponding PES
- **npes** number of PES. The parameters for the second PES are only asked if npes = 2

```
class HarmonicOscillator1D(Model):
    """ Model for a 1D harmonic oscillator with 1 or 2 potential energy surfaces """

    _questions = """
    e0 = 0.0 :: float
    w0 = 1.0 :: float
    x0 = 0.0 :: float
    # Number of potential energy surfaces
    npes = 1 :: str ::

    [npes(1)]

    [npes(2)]
    e1 = 1.0 :: float
    w1 = 1.0 :: float
    x1 = 1.0 :: float
    """

    implemented = ["energy", "gradient"]
    masses = [1.0]
    crd = [0.0]
    frequencies = [1.0]
    displacements = [[1.0]]
    modes = [Mode(freq, dis) for freq, dis in zip(frequencies, displacements)]

    @classmethod
    def from_config(cls, config):
        e0 = config['e0']
        w0 = config['w0']
        x0 = config['x0']
        npes = int(config['npes'].value)

        config_npes = config['npes']
        return cls(e0, w0, x0, npes, config_npes)

    def __init__(self, e0, w0, x0, npes, config_npes):
        self.frequencies = [w0]
        self.crd = [x0]
        self.npes = int(npes)
        self.w = [w0]
        self.x = [x0]
        self.e = [e0]

        if self.npes == 2:
            self.w += [config_npes['w1']]
            self.x += [config_npes['x1']]
            self.e += [config_npes['e1']]
```

According to the Colt style, all arguments in the main question block are given in the init method explicitly, whereas other question blocks are passed as config of the block. This makes it easy to initialize the class without user input and use it either within the framework or as a standalone.

Step 3: Implement *_energy* and *_gradient* function

The next step is to implement the actual model, i.e. the properties of the model. In our case we want to provide the energies and the gradients of the surfaces and thus private methods for *_energy* and *_gradient* are implemented.

```python
class HarmonicOscillator1D(Model):
""" Model for a 1D harmonic oscillator with 1 or 2 potential energy surfaces """

  ...

    def _energy(self, x):
        energy = []
        for i in range(self.npes):
            energy += [0.5*self.w[i]*(x - self.x[i])**2 + self.e[i]]
        energy = np.array(energy).flatten()
        return energy

    def _gradient(self, x):
        gradient = {}
        for i in range(self.npes):
            gradient[i] = np.array(self.w[i]*(x - self.x[i]))
        return gradient
```

Implement energy and gradient function

The next step is to implement the actual model. In our case we want to provide the energies and the gradients of the surfaces and thus we provide private helper functions for this: namely *_energy* and *_gradient*.


```python
class HarmonicOscillator1D(Model):
""" Model for a 1D harmonic oscillator with 1 or 2 potential energy surfaces """

  ...

    def _energy(self, x):
        energy = []
        for i in range(self.npes):
            energy += [0.5*self.w[i]*(x - self.x[i])**2 + self.e[i]]
        energy = np.array(energy).flatten()
        return energy

    def _gradient(self, x):
        gradient = {}
        for i in range(self.npes):
            gradient[i] = np.array(self.w[i]*(x - self.x[i]))
        return gradient
```


The *_energy* function takes a coordinate position, i.e. a numpy array and returns an array with one or two entries, depending on whether the surface contains one or two potential energy surfaces.

The energy is calculated according to the formula

$$E = 0.5 \cdot \omega \cdot (x - x_0)^2$$

The *_gradient* function takes a coordinate position, i.e. a numpy array and returns a dictionary. The keys in the dictionary are the state numbers as integers, i.e. 1 or 2 and the values are the gradients of the corresponding state. In our specific case, such a dictionary may look like: {1: [0.5], 2: [0.2]}

Implement get

In PySurf every computation is performed based on a user based request, that contains the properties that need to be computed. The *get* function is called with the *request* as parameter. It has to fill the request object with the desired results from the model and return it.

```
class HarmonicOscillator1D(Model):
""" Model for a 1D harmonic oscillator with 1 or 2 potential energy surfaces """

...

def get(self, request):
    """the get function returns the adiabatic energies as well as the
       gradient at the given position crd. Additionally the masses
       of the normal modes are returned for the kinetic Hamiltonian.
    """
    crd = request.crd
    print('crd', crd)
    for prop in request:
        if prop == 'energy':
            request.set('energy', self._energy(crd))
        if prop == 'gradient':
            request.set('gradient', self._gradient(crd))
    return request
```

With this we have created a very simple harmonic oscillator model.

2.2 Tutorial: Write an Ab-Initio Interface

Abinitio-Interfaces are *Plugins* for the *SurfacePointProvider* and are used to perform (electronic structure) calculations on molecular systems (NAtoms, 3). In the following example implementations are presented for a basic adapter class to the atomic simulation environment (ASE)[5] and a basic interface to PySCF.[6]

2.2.1 Example: Adapter to the ASE

The ASE is according to its documentation *a set of tools and Python modules for setting up, manipulating, running, visualizing and analyzing atomistic simulations* under GNU LPGL licence. Hereby, the ASE provides so-called *Calculators*, which are similar to the Abinitio-Interfaces used in the *SurfacePointProvider*. In the following, we are going to construct an adapter class to use the ASE calculators within our framework.

Basic Setup

To implement an Abinitio-Interface using the ASE we need to import some objects/methods. We will use from the ASE:

- *Atoms*: which defines the basic molecule
- *calculators*: which gives access to the available calculators

Additionally, we will import the Abinitio base class from Pysurf.

```
1 from ase import Atoms
2 from ase import calculators
3 from pysurf import Abinitio
```

Abstractmethods in Abinitio

```
class ASEInterface(Abinitio):
    implemented = ['energy', 'gradient']

    @classmethod
    def from_config(cls, config, atomids, nstates):
        """used to initialize the class using userinput"""

    def get(self, request):
        """Compute requested properties and set results"""
```

The *Abinitio* base class defines two abstract methods and one property that need to be set:

- **from_config:** used to initialize the class using the userinput
- **get:** which is used to answer the request
- **implemented:** states which properties are implemented, the writer of the plugin is responsible for correctness

If you write a new interface for an electronic-structure software, those are the methods you have to implement.

The next thing is to add user input.

Adding user input for the Plugin

Before we are going to implement the abstract methods we are going to add some custom user-input that we want to use in our *Plugin*. Herefor, we use the *question* DSL of Colt and add our own questions to our class.

```
class ASEInterface(Abinitio):

    _questions =  """
    calculator = qchem

    [calculator(qchem)]
    method = b3lyp
    basis = 6-31g

    [calculator(psi4)]
    method = b3lyp
    memory = 500MB
    basis = 6-31g
    """

    implemented = ['energy', 'gradient']

```

```
18     @classmethod
19     def from_config(cls, config, atomids, nstates):
20         """used to initialize the class using userinput"""
21         if nstates != 1:
22             raise Exception("ASE does not support excited states")
23         return cls(config['calculator'], atomids)
24 
25     def __init__(self, calculator, atomids):
26         # define the molecule in a basic manner
27         self.molecule = Atoms(numbers=atomids)
28         self.calculator = self._select_calculator(calculator)
29 
30     def _select_calculator(self, calculator):
31         if calculator == 'psi4':
32             return calculators.psi4.Psi4(atoms=self.molecule, method=calculator['method'],
33                                          memory=Calculator['memory'], basis=calculator['basis'])
34         if calculator == 'qchem':
35             return calculators.qchem.QChem(atoms=self.molecule, method=calculator['method'],
36                                            basis=calculator['basis'])
37         raise NotImplementedError("calculator not implemented")
```

For education purpose we only show two calculators, the one for Q-Chem and the one for Psi4 software.

Implementing get

Now it is straight-forward to implement the get function

```
1  class ASEInterface(Abinitio):
2      ...
3 
4      def get(self, request):
5          """Compute requested properties and set results"""
6          # set the coordinates
7          self.molecule.positions = request.crd
8          # compute energy
9          if 'energy' in request:
10             request['energy'].set(self.calculator.get_potential_energy())
11         # compute gradient
12         if 'gradient'in request:
13             request['gradient'].set(self.calculator.get_forces())
14         return request
```

With that done, we have a new Abinitio-Interface

2.2.2 Example: PySCF interface:

In this example we show how to write an interface for the PySCF program package, which also supports excited states.

Step 0: Basic Setup

To implement an Abinitio-Interface using PySCF we need to import some objects/methods. We will use from the PySCF

- *gto*: which defines the basic molecule
- *dft*, *grad*, *tddft*: which allow to perform dft and tddft calculations for energies and gradients

From PySurf we will import the Abinitio class which is the base class of all Abinitio interfaces

```
from pysurf import Abinitio
from pyscf import gto, dft, tddft, grad
```

Step 1: Abstract methods in Abinitio

```
class PySCF(Abinitio):

    methods = {}
    implemented = []

    @classmethod
    def from_config(cls, config, atomids, nstates):
        """used to initialize the class using userinput"""

    def get(self, request):
        """Compute requested properties and set results"""
```

The *Abinitio* base class defines two abstract methods that need to be set as well as one property:

- **from_config:** used to initialize the class using the userinput
- **get:** which is used to answer the request
- **implemented:** states which properties are implemented

The next thing is to add user input.

Step 2: Adding user input for the Plugin

Before we are going to implement the abstract methods we are going to add some custom user-input that we want to use in our *Plugin*. Herefor, we use the *question* DSL of Colt and add our own questions to our class.

For each electronic-structure method of PySCF a separate class is implemented, the calculator classes. These calculator classes have to have methods with the name *do_prop* where prop stands for all the implemented properties, e.g. *do_energy*. Moreover it has to have a property *implemented* which is copied to the PySurf class. PySurf will check the *implemented* property, whether the interaface provides all necessary properties that are needed in the calculation.

```
class PySCF(Abinitio):

    _questions =  """
    basis = 631g*
    # Calculation Method
    method = DFT/TDDFT :: str :: [DFT/TDDFT]
    """

    # implemented has to be overwritten by the individual classes for the methods
    implemented = []

    # dictionary containing the keywords for the method and the corresponding classes
    methods = {'DFT/TDDFT': DFT}

    @classmethod
    def _extend_questions(cls, questions):
        questions.generate_cases("method", {name: method.questions
                                             for name, method in cls.methods.items()})

    @classmethod
    def from_config(cls, config, atomids, nstates):
        method = config['method'].value
        basis = config['basis']
        config_method = config['method']
        return cls(basis, method, atomids, nstates, config_method)


    def __init__(self, basis, method, atomids, nstates, config_method):
        """ """
        self.mol = self._generate_pyscf_mol(basis, atomids)
        self.nstates = nstates
        self.atomids = atomids
        self.basis = basis
        # initializing the class for the corresponding method
        self.calculator = self.methods[method].from_config(config_method, self.mol, nstates)
        # update the implemented property
        self.implemented = self.calculator.implemented
```

The code for the *_generate_pyscf_mol* function is shown in the next section. It is a PySCF specific function that creates the molecule object for PySCF.

Step 3: Implementing get

The get function calls the corresponding functions of the calculator class. The *_generate_pyscf_mol* function generates the basic molecule object of Pyscf.

```
class PySCF(Abinitio):
    ...
    def get(self, request):
        # update coordinates
        self.mol = self._generate_pyscf_mol(self.basis, self.atomids, request.crd)
        for prop in request:
            func = getattr(self.calculator, 'do_' + prop)
            func(request, self.mol)
        #
        return request

    @staticmethod
    def _generate_pyscf_mol(basis, atomids, crds=None):
        """ helper function to generate the mol object for Pyscf """
        if crds is None:
            crds = np.zeros((len(atomids), 3))
        mol = gto.M(atom=[[atom, crd] for atom, crd in zip(atomids, crds)],
        basis = basis, unit='Bohr')
        return mol
```

Step 4: Implementing the DFT calculator class

For educational purposes we restrict ourselves to the calculation of energies. Like in all *Colt* classes questions can be added, which are asked through the *_extend_questions* method of the *PySCF* class. Answers are passed to the *__init__* function via the *from_config* classmethod. At the initialization the dft and tddft scanners are set up to make sure that calculations are started from the last converged result. In the *do_energy* function the request is filled with the energies.

```
class DFT(Colt):
    """ class which executes the DFT and TDDFT calculations using the PySCF package """

    _questions = """
    functional = :: str :: ['pbe0']
    basis = ccpvdz :: str
    """

    implemented = ['energy']


    @classmethod
    def from_config(cls, config, mol, nstates):
        """ """
        functional = config['functional']
        return cls(functional, mol, nstates)


    def __init__(self, functional, mol, nstates):
        self.mol = mol
        self.nstates = nstates

        mydft = dft.RKS(mol).x2c().set(xc=functional)
        self.dft_scanner = mydft.as_scanner()
```

```
        if self.nstates > 1:
            mydft._numint.libxc = dft.xcfun
            mytddft = tddft.TDDFT(mydft)
            self.tddft_scanner = mytddft.as_scanner()
            self.tddft_scanner.nstates = self.nstates - 1


    def do_energy(self, request, mol):
        if self.nstates == 1:
            en = [self.dft_scanner(mol)]
        else:
            en = self.tddft_scanner(mol)

        request.set('energy', en)
```

2.3 Tutorial: Write an Interpolator

In case you want to extend the interpolation facilities you can write an additional **Interpolator**.

First, it has to be decided what interpolator should be implemented. In this tutorial a trivial nearest neighbor interpolator is implemented. A Nearest Neighbor Interpolator just looks to the points closest to the point we want to interpolate. Then, the properties of this nearest dataset are returned for the requested coordinates. Our implementation in this tutorial will use scipy's nearest neighbor search algorithm for a fast search and overall performance.

2.3.1 Example: Nearest Neighbor Interpolator

Interpolator inherite from the *Interpolator* base class. The base class takes care of the basic logic, which are common to any interpolator, e.g. coordinate transformation and energy-only calculations etc. During the example we will encounter a few of them and explain them, whenever they appear.

```
from scipy.spatial import cKDTree
from pysurf.spp import Interpolator
```

Basic Features

Every **Interpolator** has to implement 5 functions:

def get(self, request): fill the request with the demanded data and additionally a flag has to be returned, whether the result is trustworthy If *fit_only* of the SPP is False, an electronic structure calculation is started, else the result of the interpolator is taken. If *fit_only* is True, the result of the interpolator will be used anyway.

def get_interpolators(self, db, properties): for each property that is requested, a separate interpolator is set up. The function has to return a dictionary that contains the property name and the interpolator for the property

def save(self, filename): The save method is primarily for machine learning algorithms. The weights have to be put in a file.

def get_interpolators_from_file(self, filename, properties): This function reads the weights-file and sets up the interpolators for the properties from the weights-file. Specifically in machine learning algorithms, it is not necessary to train the algorithm again.

def _train(self): This function uses the input data to train the interpolator. It is primarily used in machine learning interpolators.

def loadweights(self, filename): The weights are loaded from a file.

```
1  class NearestNeighborInterpolator(Interpolator):
2      """Nearest Neighbor Interpolator"""
3
4      @classmethod
5      def from_config(cls, config, db, properties, logger, energy_only, weightsfile, crdmode,
6                      fit_only):
7          """ This classmethod overwrites the from_config method of the factory and is needed
8              if interpolator specific user input is implemented
9          """
10
11     def __init__(self, db, properties, logger, energy_only=False, weightsfile=None,
12                  crdmode='cartesian', fit_only=False)
13         """ This init overwrites the init of the interpolator factory. Therefor, it is
14             advisable to call the init of the interpolator factory to make sure that all
15             functionality is working and only additional features are added here.
16         """
17         super().__init__(db, properties, logger, energy_only, weightsfile, crdmode=crdmode,
18                          fit_only=fit_only)
```

```
19
20
21     def get(self, request):
22         """fill request
23
24            Return request and if data is trustworthy or not
25         """
26
27     def get_interpolators(self, db, properties):
28         """ """
29
30     def save(self, filename):
31         """Save weights"""
32
33     def get_interpolators_from_file(self, filename, properties):
34         """setup interpolators from file"""
35
36     def _train(self):
37         """train the interpolators using the existing data"""
38
39     def loadweights(self, filename):
40         """load weights from file"""
```

User Input

PySurf is build around the Colt framework, developed independently. To specify custom user input needed for your class you simply use the *_questions* string: In our example we will need 4 user inputs:

- **trust_radius_general, float** the radius to decide whether an interpolation is trustworthy
- **trust_radius_ci, float** the radius in the region of small energy gaps to decide whether an interpolation is trustworthy
- **energy_threshold, float** the threshold to distinguish between regions with small and large energy gap
- **norm, str** the norm that is used to measure the distance between points

```
1  class NearestNeighborInterpolator(Interpolator):
2      """Basic Rbf interpolator"""
3
4      _questions = """
5          trust_radius_general = 0.75 :: float
6          trust_radius_ci = 0.25 :: float
7          energy_threshold = 0.02 :: float
8          norm = euclidean :: str :: [euclidean]
9      """
10     @classmethod
11     def from_config(cls, config, db, properties, logger, energy_only, weightsfile, crdmode, fit_only):
12         trust_radius_general = config['trust_radius_general']
13         trust_radius_CI = config['trust_radius_ci']
14         energy_threshold = config['energy_threshold']
15         #
16         # convert input for norm in corresponding input (p-Norm) for the cKDTree
17         # for more information go to the cKDTree.query documentation
```

```
        if config['norm'] == 'manhattan':
            norm = 1
        elif config['norm'] == 'max':
            norm = 'infinity'
        else:
            norm = 2
        #
        return cls(db, properties, logger, energy_only=energy_only, weightsfile=weightsfile,
                   crdmode=crdmode, trust_radius_general=trust_radius_general,
                   trust_radius_CI=trust_radius_CI, energy_threshold=energy_threshold,
                   fit_only=fit_only, norm=norm)

    def __init__(self, db, properties, logger, energy_only=False, weightsfile=None,
                 crdmode='cartesian', fit_only=False, trust_radius_general=0.75,
                 trust_radius_CI=0.25, energy_threshold=0.02, norm='euclidean'):
        self.trust_radius_general = trust_radius_general
        self.trust_radius_CI = trust_radius_CI
        self.energy_threshold = energy_threshold
        self.tree = None
        self.norm = norm
        # Call the init method of the Interpolator Factory
        super().__init__(db, properties, logger, energy_only, weightsfile,
                         crdmode=crdmode, fit_only=fit_only)
```

Parameters

db: databse containing the datasets, on which the interpolation is based on

properties: list properties (e.g. [‘energy’, ‘gradient’]) that should be fitted

logger: logger to log any incident

energy_only: bool, optional if energy_only is True, gradients are derived from the energy surface

weightsfile: str, optional filepath, where to save the weights. Not used in the case of the NearestNeighborInterpolator, but needed for the overall framework.

crdmode: str, optional Variable to determine whether a coordinate transformation is applied before fitting.

fit_only: bool, optional Flag to determine, whether no new QM calculations are performed

trust_radius_general: float, optional radius to determine whether fitted result is trustworthy in regions of a large energy gap

trust_radius_CI: float, optional radius to determine whether fitted result is

trustworthy in regions of a small energy gap

energy_threshold: float, optional Threshold to distinguish regions of small and large energy gaps.

norm: str, optional Determining the norm for the nearest neighbor search. 'manhattan' corresponds to the 1-norm, 'euclidean' is the 2-norm, and 'max' is the infinity norm.

Implement *get_interpolators* function

The next step is to implement the *get_interpolators* method and the helper class for the NearestNeighborInterpolator of each property *NNInterpolator*. For each property, an Interpolator is set up, which is an instance of the *NNInterpolator* class. Each interpolator has to be callable and to return the desired property.

```
1  class NearestNeighborInterpolator(Interpolator):
2      """Nearest Neighbor Interpolator"""
3
4      ...
5        def get_interpolators(self, db, properties):
6            """ """
7            self.tree = cKDTree(self.crds)
8            return {prop_name: NNInterpolator(db, self.tree, prop_name)
9                    for prop_name in properties}, len(db)
10
11
12  class NNInterpolator():
13      def __init__(self, db, ckdtree, prop):
14          self.db = db
15          self.tree = ckdtree
16          self.prop = prop
17
18      def __call__(self, crd, request=None, idx=None):
19          if idx is None:
20              dist, idx = self.tree.query(crd)
21          return self.db.get(self.prop, idx)
```

The *get_interpolators* method returns a dictionary with the property names as keys and the interpolator for that specific property as value. For each property a separate interpolator has to be set up so that the interpolator factory can handle the interpolators for the properties independently, which allows e.g. the *energy_only* calculations. Implementing the interpolators in this way, they naturally are included in the code package and the full functionality is available.

To avoid that the cKDTree is set up several times, the *NNInterpolator* takes the tree as a Parameter. Moreover, if *NNInterpolator* is called with an index, no nearest neighbor search is performed, but the property of the dataset with the index is returned. This is important in the case when several properties are demanded so that the nearest neighbor search is done only once, cf. Step 4 and the *get* function.

Implement *get* function

The *get* function is called with the *request* as parameter. It has to fill in the desired results from the fit into the *request* instance and state whether the fit is trustworthy.

```
class NearestNeighborInterpolator(Interpolator):
    """Nearest Neighbor Interpolator"""

    ...

 def get(self, request):
        #
        # Convert coordinate into desired format
        if self.crdmode == 'internal':
            crd = internal(request.crd)
        else:
            crd = request.crd
        #
        # Make nearest neighbor search once and pass it to all interpolators
        dist, idx = self.tree.query(crd, p=self.norm)
        for prop in request:
            request.set(prop, self.interpolators[prop](crd, request, idx))
        #
        # Determine whether result is trustworthy, using the trust radii
        diffmin = np.min(np.diff(request['energy']))
        is_trustworthy = False
        if diffmin < self.energy_threshold:
            if dist < self.trust_radius_CI: is_trustworthy = True
        else:
            if dist < self.trust_radius_general: is_trustworthy = True
        #
        return request, is_trustworthy
```

The *get* function first has to make sure that the interpolators get the right coordinates. Subsequently, the interpolators for all the desired properties are called and the results are put into the request instance. Finally, it is checked, whether the requested point is within the trusted region. The trusted region is devided into two parts, depending whether the smallest energy gap between two potential energy surfaces is small or large. The threshold is given as the *energy_threshold* as user input as well as the radii *trust_radius_ci* and *trust_radius_general*.

Implement the save, load and _train methods

The NearestNeighborInterpolator does not use a save and load function, which can be used to store data in a file and read afterwards, to avoid multiple training sessions. For this particular case, the training of the Nearest Neighbor interpolator is just the update of the cKDTree. Therefor, these functions are not really used, but implemented in a way to make sure that the full functionality is available.

```
class NearestNeighborInterpolator(Interpolator):
   """Nearest Neighbor Interpolator"""

   ...

     def loadweights(self, filename):
          """ Weights are loaded for the interpolators from a file. As the
              NearestNeighborInterpolator is not using the save option, also
              here, interpolators are just set up from the database

              Parameters:
              -----------
                  filename, str:
                      filepath of the file containing the weights. Not used here!
          """
          #
          self.logger.warning("NearestNeighborInterpolator cannot load weights, interpolators are " +
                              "set up from DB")
          # As saving is not used, interpolators are set up from the database
          self.get_interpolators(self.db, self.properties)

     def save(self, filename):
         """ Method to save the interpolators to a file. Not used here!

             Parameters:
             -----------
                 filename:
                     filepath where to save the information. Not used here!
         """
         #
         self.logger.warning("NearestNeighborInterpolator cannot be saved to a file")

     def _train(self):
         """ Method to train the interpolators. In the case of the NearestNeighborInterpolator
             the cKDTree has to be updated.
         """
         #update cKDTree
         self.tree = cKDTree(self.crds)
```

3 Workflow

3.1 Tutorial: How to use the Workflow engine

In this tutorial we will show how to perform an ab-initio single point calculation using the *PySurf* framework. If you have any other electronic structure program installed that is supported by PySurf, you can just use it. Otherwise you can download the *PySCF* program package and install it via:

```
pip install pyscf
```

PySCF is free of charge. In this tutorial we use it to do a TDDFT calculation.

To start you need a folder for the calculation and an xyz file of your molecule. There is an example ("<pysurf>/examples/so2/so2.xyz") of a SO_2 geometry. Go to your favorite place and make a folder for the test calculation and copy your coordinate file:

```
mkdir test_so2
cd test_so2
cp <so2.xyz> ./
```

<so2.xyz> stands for the path of your SO_2 coordinate file.

Finally you have to run the *sp_calc.py* workflow for a single point calculation. <pysurf> is the path where your PySurf package is installed

```
python <pysurf>/bin/sp_calc.py so2.xyz 2 energy
```

The *sp_calc.py* script takes three positional arguments. The first is the path of the coordinate file, the second is the number of states and the third is a list of the properties that you want to calculate. If you also want to have the gradients you have to write:

```
python <pysurf>/bin/sp_calc.py so2.xyz 2 "energy, gradient"
```

For help text of the script type:

```
python <pysurf>/bin/sp_calc.py -h
```

Subsequently PySurf will guide you through all the questions of the *SurfacePointProvider* and the electronic structure interface. You can take the defaults or choose your own. The *Colt* framework helps you that your answers are consistent. In this case, we accept all the defaults up to the point for the *software*. There we choose *PySCF*. Subsequently we take the defaults again. The *PySCF* calculation is started and the output is printed to the screen. The results are saved in the database that was put in the input (the default is just *db.dat*). To see all the parameters of your calculation, you can open the *spp.inp* file, where the whole input is stored. If you want to repeat the calculation with the same settings for a different geometry, you can just exchange the coordinate file. Typing again:

```
python <pysurf>/bin/sp_calc.py so2.xyz 2 "energy, gradient"
```

will start a new calculation. This time no questions will be asked, but all the answers are read from the *spp.inp* file.

Congratulations, you just performed your first single point calculation with PySurf!

3.2 Tutorial: Write your own Workflow

Every scientist is after their own ideas and visions to move the boarders of knowledge. Therefor, it is natural that predefined tools can never be sufficient for all tasks that scientists want to do. The PySurf Workflow engine provides a toolbox of nodes, which can be combined like “Lego Bricks” to new powerful algorithms. If your desired functionality is not yet in the toolbox, you can easily add it and you can of course include the full functionality of the Workflow nodes in your Python scripts. The Workflow engine comes with its own domain specific language to combine all different nodes like in a normal script. The engine checks that the input and output of the nodes fit together. Let’s see how it works. As an example we take the workflow of the single point calculation, which you just performed in the previous section.

3.2.1 Workflow framework

For the Workflow framework, the *engine* has to be imported from *pysurf.workflow*. With the command *engine.create_workflow()* the workflow is generated. The first argument is an arbitrary name given to the workflow, the second argument is a multiline string which contains the workflow. With the command *workflow.run()* the workflow is executed.

```
from pysurf.workflow import engine


workflow = engine.create_workflow("populations", """
...
""")

wf = workflow.run()
```

3.2.2 Single Point Calculation as Workflow

Here, the Workflow for a single point calculation is shown.

```
workflow = engine.create_workflow("sp_calc", """
crd = read_xyzfile_crd(crd_file)
atomids = read_xyzfile_atomids(crd_file)
spp = spp_calc("spp.inp", atomids, nstates, properties=properties)
res = sp_calc(spp, crd, properties=properties)
""")
```

- **read_xyzfile_crd:** node that returns the xyz coordinates from a xyz file
- **read_xyzfile_atomids:** node that returns the atomids from a xyz file
- **spp_calc:** node that initializes a SPP using an inputfile (*"spp.inp"*), atomids (as integer list), number of states (as integer) and the desired properties (as list)
- **sp_calc:** node that sends the request to an initialized SPP. To start a calculation it is important that the SPP has been initialized with *spp_calc* and not *spp_analyse*. The second uses interpolation to produce the results. As arguments it takes the SPP (*spp*), the coordinates (*crd*) and the properties (*list*)

Variables which are not defined within the workflow are asked via the command line. In this case the user has to specify the xyz file (*crd_file*), the number of states (*nstates*) and the properties that

should be calculated as list, e.g. if you want to calculate energies, gradients and oscillator strengths you have to pass the list "energy, gradient, fosc"

3.2.3 Using the results

All variables of a workflow are saved in a dictionary. Either results are used and processed within the workflow, or you can read them from the dictionary and use them in your own script.

```
workflow = engine.create_workflow("populations", """
...
res = sp_calc(spp, crd, properties=properties)
""")

wf = workflow.run()
print(wf['res']['energy'])
```

3.2.4 Appendix: How to put data into the Workflow

It is not only possible to take data out of the workflow, but also to put it in via a dictionary when the workflow is executed.

```
workflow = engine.create_workflow("populations", """
...
""")

wf = workflow.run({"properties": ['energy']})
```

References

(1) Müller, H.; Köppel, H. Adiabatic wave-packet motion on conically intersecting potential energy surfaces. The case of $SO_2(^1B_1^1A_2)$. *Chem. Phys.* **1994**, *183*, 107 – 116.

(2) Lévêque, C.; Komainda, A.; Taïeb, R.; Köppel, H. Ab initio quantum study of the photodynamics and absorption spectrum for the coupled 1^1A_2 and 1^1B_1 states of SO_2. *J. Chem. Phys.* **2013**, *138*, 044320.

(3) Mai, S.; Marquetand, P.; González, L. Non-adiabatic and intersystem crossing dynamics in

SO_2. II. The role of triplet states in the bound state dynamics studied by surface-hopping simulations. *J. Chem. Phys.* **2014**, *140*, 204302.

(4) Plasser, F.; Gómez, S.; Menger, M. F. S. J.; Mai, S.; González, L. Highly efficient surface hopping dynamics using a linear vibronic coupling model. *Phys. Chem. Chem. Phys.* **2019**, *21*, 57–69.

(5) Larsen, A. H. et al. The atomic simulation environment-a Python library for working with atoms. *J. Phys.: Condens. Matter* **2017**, *29*, 273002.

(6) Sun, Q.; Berkelbach, T. C.; Blunt, N. S.; Booth, G. H.; Guo, S.; Li, Z.; Liu, J.; McClain, J. D.; Sayfutyarova, E. R.; Sharma, S.; Wouters, S.; Chan, G. K.-L. PySCF: the Python-based simulations of chemistry framework. *WIREs Comput. Mol. Sci.* **2018**, *8*, e1340.