CONTENTS

1 About 1
2 Citation 3
3 Indices and tables 43
Python Module Index 45
Index 47
BasisOpt is an open-source library for the optimization of gaussian basis sets, for use in quantum chemistry calculations. We aim to offer:

- a software-agnostic interface to allow for reproducibility
- automation of the optimization procedure
- methods for evaluating and visualising basis sets
- a framework for the development of new optimization methods
- standardised format for presenting basis set optimization results

It was written predominantly by Robert Shaw (@robertshaw383), with assistance from Grant Hill (@Dr_GHill), as part of an EPSRC-funded project at the University of Sheffield.
If you make use of this software, please consider citing the following:
[ChemRxiv preprint](https://chemrxiv.org/engage/chemrxiv/article-details/640f48e3b5d5dbe9e832e997)

## 2.1 Installation

The easiest way to install is via pip:

```bash
pip install basisopt
```

### 2.1.1 Install from Source

First, make sure you install the python package `poetry`. Then, clone the repository from `github`:

```bash
git clone https://github.com/robashaw/basisopt.git
cd basisopt
poetry install -v
```

Then, an interactive shell or script can be run with:

```bash
poetry run [python3 or script.py]
```

Alternatively, if you wish to use your locally-changed package elsewhere, then try the following with `conda`, for example, in the top-level directory:

```bash
conda create -n basisopt python=3.10 pip install -e .
```

### 2.1.2 Dependencies

There are fairly minimal requirements, and versions listed are minimum tested versions. Older versions _may_ work but are untested. The exception is Mendeleev where upstream changes mean we currently require version 0.9; this will hopefully be fixed soon.

- python >= 3.9
- colorlog >= 4.1
- numpy >= 1.21.6
BasisOpt

- scipy >= 1.8.1
- matplotlib >= 3.3
- pandas >= 1.3.5
- monty >= 2022.4.26
- mendeleev == 0.9.0
- basis_set_exchange >= 0.9

You will also need a quantum chemistry backend. Currently supported codes are:
- Psi4 : version 1.4 or above
- ORCA : version 4.2.x or version 5.x

There will eventually be a tutorial on how to implement wrappers for different packages. If you do, please consider submitting a pull request so that others may benefit.

2.1.3 Testing and development

We use pytest for our CI:

```
poetry run pytest
```

We also use black, isort, and flake8, for code-formatting and linting. Any PR will automatically be checked using these packages before being considered for merging.

2.1.4 Documentation

To build this documentation locally, make sure you have the following packages

- Sphinx >= 2.1.2

In the doc directory, run the following:

```
mkdir build
sphinx-build src/ build/
```

This will by default create HTML; to make other formats, consult the sphinx-build documentation.

2.2 Quick start

The basic components in a BasisOpt workflow are:

- calculation backend
- molecules
- basis sets
- optimization strategies

The tutorials section gives more detailed explanations on how to perform various types of basis set optimization. This quick start guide is intended to go through the steps necessary to get ready for an optimization.
2.2.1 Load a backend

Currently there are two options for quantum chemistry backend, although any new wrappers will follow the same structures. The first steps upon importing BasisOpt should always be

1. select a backend
2. set the scratch directory (defaults to current directory)

Optionally you may also want to
3. change the logging settings
4. enable/disable parallel calculations

For Psi4, the python API is used:

```python
import basisopt as bo
bo.set_backend('psi4')
bo.set_tmp_dir('/tmp/')
```

For ORCA (and other non-native backends), we need to give the path to the directory containing the executables, e.g.

```python
bo.set_backend('orca', path='/usr/local/bin/orca/
bo.set_tmp_dir('workdir/')
```

Internally, this will check that the backend is usable. If this is successful, you will get confirmation:

```
```

Otherwise you will get an error:

```
2022-09-11 21:23:17,502 - psi4 - ERROR - Psi4 backend not found!
```

The “Dummy” backend allows BasisOpt to be used for analysis/vizualisation, but cannot perform calculations.

To change the level of logging (logging.INFO by default), and enable parallelism using DASK (disabled by default):

```python
import logging
bo.set_logger(level=logging.WARNING, filename="bo.log")
bo.set_parallel(True)
```

2.2.2 Create a molecule

Molecules have four basic properties, before a basis set is added:

- name
- charge (default 0)
- spin multiplicity (default 1)
- coordinates

Having a unique name field is important when running calculations over multiple molecules, as it acts as an identifier.

There are three routes to creating a molecule:

1. create an empty molecule and add atoms manually
m = bo.Molecule(name="Chlorine", charge=-1, mult=1)
m.add_atom(element='Cl', coord=[0., 0., 0.])

2. load from an XYZ file

m = bo.Molecule.from_xyz("water.xyz", name="Water",
charge=0, mult=1)

3. create a diatomic from a string (e.g. nitric oxide with a bond distance of 1.3 angstrom)

from basisopt.molecule import build_diatomic
m = build_diatomic("NO,1.3", charge=0, mult=2)

2.2.3 Dummy atoms

You can set certain atoms to be dummies (or ghosts) by specifying the indices of the atoms you want to have no electrons:

```python
# the first three atoms will be marked as dummies
m.set_dummy_atoms([0, 1, 2])
```

This can be useful for example in the development of midbond functions, where you might wish to calculate interaction energies.

2.2.4 Add a basis set

Basis sets internally in BasisOpt are dictionaries with the following structure:

```python
basis = {
    'H': [s-Shell, p-Shell, ...],
    'O': [s-Shell, p-Shell, ...],
    ...
}
```

where basisopt.containers.Shell objects have three properties:

- angular momentum (‘s’, ‘p’, ‘d’, …)
- exponents (numpy array)
- coefficients (list of numpy arrays with same length as exponents)

These can be created manually, fetched directly from the basis set exchange, or read in from file using the basis set exchange API. For example:

```python
# fetch from BSE library
m.basis = bo.fetch_basis('cc-pvdz', ['H', 'O'])

# load from file
import basis_set_exchange as bse
from basisopt.bse_wrapper import bse_to_internal
bse_basis = bse.read_formatted_basis_file('vdz.basis', basis_fmt='molpro')
m.basis = bse_to_internal(bse_basis)
```
2.2.5 Add ECPs

If you want ECPs to be used on particular atoms, this can be specified by providing a dictionary of ECP names:

```python
m.set_ecps(
    {'Br': 'aug-cc-pvtz-pp'}
)
```

If you are using Psi4, these are looked up from the basis set exchange, so should match names given there. If you are using Orca, the internal Orca ECPs are used, a list of names for which can be found in Section 6.3.3 of the Orca manual.

2.2.6 Running a calculation

To test that everything is set up correctly, you can run a quick calculation as follows:

```python
m.method = 'hf'
success = bo.run_calculation(evaluate='energy', mol=m)
print(bo.get_backend().get_value('energy'))
```

2.3 Tutorials

- _tutorials/eventemper
- _tutorials/molecule
- _tutorials/ranking
- _tutorials/multimol
- _tutorials/reduce
- _tutorials/visualize

2.4 BasisOpt API

2.4.1 basisopt package

Subpackages

basisopt.basis package

Submodules

basisopt.basis.atomic module

```python
class basisopt.basis.atomic.AtomicBasis(name: str = 'H', charge: int = 0, mult: int = 1)
```

Bases: Basis

Object for preparation and optimization of a basis set for a single atom.
et_params
    even tempered expansion parameters
    Type
data.ETParams
charge
    net charge on atom
    Type
    int
multiplicity
    spin multiplicity of atom
    Type
    int
config
    configuration of basis, (k, v) pairs of form (angular momentum: no. of functions) e.g. ‘s’: 5, ‘p’: 4, etc.
    Type
dict

Special attribute:
    element: gets Mendeleev Element object of atom (_element)
    set with atomic symbol

Private Attributes:
    _element (mendeleev Element): object set via element_molecule (Molecule): Molecule object rep of atom
    _done_setup (bool): flag for whether ready for optimize
    _symbol (str): atomic symbol in lowercase

as_dict() → dict[str, Any]
    Returns MSONable dictionary of AtomicBasis

as_xyz() → str
    Returns the atom as an xyz file string

property charge: int

property config: dict[str, int]

configuration(quality: str = 'dz')
    Sets the basis set configuration to a desired quality
    Parameters
    quality (str) – name of quality type, see zetatools for options

contract()
    Handles contraction of primitives

property element: element

classmethod from_dict(d: dict[str, Any]) → object
    Creates an AtomicBasis from MSONable dictionary

minimal() → dict[str, int]
    Returns the minimal basis configuration for this atom
**property multiplicity**: int


Runs the basis optimization

**Parameters**

- **algorithm**(str) – optimization algorithm to use, see scipy.optimize for options
- **params**(dict) – dictionary of parameters to pass to the backend - see the relevant Wrapper object for options

**Returns**

a dictionary of scipy results from each opt step

**Return type**

opt_results (OptResult)

**save**(filename: str)

Pickles the AtomicBasis object into a binary file

**set_even_tempered**(method: str = 'hf', accuracy: float = 1e-05, max_n: int = 18, max_l: int = -1, exact_ref: bool = True, params: dict[str, Any] = {})

Looks up or computes an even tempered basis expansion for the atom

**Parameters**

- **method**(str) – method to use; possibilities can be found through Wrapper object
- **accuracy**(float) – the tolerance to optimize to, compared to reference value
- **max_n**(int) – max number of primitives per shell
- **max_l**(int) – angular momentum to go up to; if -1, will use max l in minimal config
- **exact_ref**(bool) – uses exact numerical HF energy if True, False (calculates cc-pV5Z reference value if) –
- **params**(dict) – dictionary of parameters to pass to the backend -
- **options**(see the relevant Wrapper object for) –

**Sets:**

self.et_params


Sets up the basis ready for optimization. Must be called before optimize is called

**Parameters**

- **method**(str) – the method to use; available methods can be checked via the Wrapper object
- **quality**(str) – configuration quality, see zetatools for options
- **strategy**(Strategy) – the optimization strategy to use, e.g. EvenTemperedStrategy
- **reference**(tuple) – the optimization strategy to use, e.g. EvenTemperedStrategy
- **(name)(basis_name, None) –
• OR (value) –

• BSE (requested basis from the) –

• params (dict) – dictionary of parameters to pass to the backend - see the relevant Wrapper object for options

Sets:
self.strategy (Strategy): optimization strategy
self.config (Configuration): basis set configuration
self._done_setup (bool): cannot call optimize until this flag is True

basisopt.basis.atomic.needs_element(func: Callable) → Callable
Decorator that checks if the AtomicBasis has an element attribute :raises ElementNotSet if no element found:

basisopt.basis.basis module

class basisopt.basis.basis.Basis
Bases: MSONable

Abstract parent class object representing a basis type. All basis types must inherit from here to work, see e.g. AtomicBasis, MolecularBasis

results
a Result object where any results (e.g. calculations, optimizations, …)

can be archived

Private attributes:
_tests (list): a list of Test objects that can be run and collated together, with results going into the results attribute

as_dict() → dict[str, Any]
A JSON serializable dict representation of an object.

copy() → object
Returns a deep copy of self.

classmethod from_dict(d: dict[str, Any]) → object

Parameters

d – Dict representation.

Returns
MSONable class.

get_basis() → dict[str, list[basisopt.containers.Shell]]

get_test(name: str) → Test | None
Retrieve a Test with a given name if it exists.

load(filename: str) → object
Loads and returns a Basis object from a binary file pickle.

All basis objects should implement an optimize method with this signature.
```python
register_test(test: Test)
    Add a Test object to the set of tests

run_all_tests(params: dict[str, Any] = {})
    Runs all the tests in basis, printing results

run_test(name: str, params: dict[str, Any] = {})
    Runs a test with the given name, printing result

save(filename: str)
    Pickles the Basis object into a binary file

basisopt.basis.basis.even_temper_expansion(params: list[tuple[float, float, int]]) -> list[basisopt.containers.Shell]
    Forms a basis for an element from even tempered expansion parameters

    Parameters
    • params (list) – list of tuples corresponding to shells
      • (e.g.) –
        • *(is expanded as c_l)* –

    Returns
    list of Shell objects for the expansion

basisopt.basis.basis.fix_ratio(exps: ndarray, ratio: float = 1.4) -> ndarray
    Returns a sorted numpy array of exponents where x_{i+1}/x_i >= ratio

basisopt.basis.basis.uncontract(basis: dict[str, list[basisopt.containers.Shell]], elements: list[str] | None = None) -> dict[str, list[basisopt.containers.Shell]]
    Uncontracts all shells in a basis for the elements specified (does not overwrite the old basis).

    Parameters
    • basis (dict) – the basis dictionary to be uncontracted
      • elements (list) – list of atomic symbols

    Returns
    a new basis dictionary with uncontracted shells

basisopt.basis.basis.uncontract_shell(shell: Shell)
    Converts a Shell into an uncontracted Shell (overwrites any existing contraction coefs)
```

### basisopt.basis.guesses module

basisopt.basis.guesses.bse_guess(atomic, params={‘name’: ‘cc-pvdz’})
    Takes guess from an existing basis on the BSE

    Params:
    • name (str): name of desired basis set

basisopt.basis.guesses.even_tempered_guess(atomic, params={})
    Takes guess from an even-tempered expansion

    Params:
    • see signature for AtomicBasis.set_even_tempered
basisopt.basis.guesses.log_normal_guess(atomic, params={"mean": 0.0, 'sigma': 1.0})
Generates exponents randomly from a log-normal distribution

Params:
  mean: centre of the log-normal distribution
  sigma: standard deviation of log-normal distribution

basisopt.basis.guesses.null_guess(atomic, params={})
Default guess type for testing, returns empty array

basisopt.basis.molecular module

class basisopt.basis.molecular.MolecularBasis(name: str = 'Empty', molecules: list[basisopt.molecule.Molecule] = [])
Bases: Basis
Object for preparation and optimization of a basis set for multiple atoms across one or more Molecules.

basis
  internal basis used for all molecules

  Type
dict

Private Attributes:
  _molecules (dict): dictionary of Molecule objects
  _atoms (set): unique atoms across all molecules
  _atomic_bases (dict): dictionary of AtomicBasis objects
    for each atom in _atoms
  _done_setup (bool): if True, setup has been called

add_molecule(molecule: Molecule)
Adds a Molecule object to the optimization pool

as_dict() → dict[str, Any]
Returns as MSONable dictionary

classmethod from_dict(d: dict[str, Any]) → object
Creates a MolecularBasis from an MSONable dictionary

get_atomic_basis(atom: str) → AtomicBasis
Returns the AtomicBasis object for a given atom, if it exists, otherwise None

get_basis() → dict[str, list[basisopt.containers.Shell]]
Returns the basis set used for all molecules

get_molecule(name: str) → Molecule
Returns a Molecule with the given name, if it exists, otherwise None

molecules() → list[basisopt.molecule.Molecule]
Returns a list of all the Molecule objects

optimize(algorithm: str = 'Nelder-Mead', params: dict[str, typing.Any] = {}, reg:
Calls collective optimize to optimize all the atomic basis sets in this basis

Parameters
• **algorithm** (*str*) – name of scipy.optimize algorithm to use
• **params** (*dict*) – parameters to pass to scipy.optimize
• **reg** (*callable*) – regularization to use
• **npass** (*int*) – number of optimization passes to do
• **parallel** – if True, molecular calculations will be distributed in parallel

```python
run_all_tests(params: dict[str, Any] = {}, reference_basis: str | dict[str, list[basisopt.containers.Shell]] | None = None) -> None
```

Runs all of the tests across all molecules, and prints the results to logger

**Parameters**
- **params** (*dict*) – parameters to pass to the backend
- **reference_basis** (*str or dict*) – either string name for basis to fetch from the BSE, or an internal basis dictionary, or None

```python
```

Runs a single test with a given name across all molecules

**Parameters**
- **name** (*str*) – name of the test
- **params** (*dict*) – parameters for backend
- **reference_basis** (*str or dict*) – either string name for basis to fetch from the BSE, or an internal basis dictionary, or None
- **do_print** (*bool*) – if True, test results will be printed to Logger

**Returns**
Dicionary of results for each test, indexed by molecule name

```python
save(filename: str)
```

Pickles the MolecularBasis object into a binary file

```python
```

Sets up the basis ready for optimization by creating AtomicBasis objects for each unique atom in the set, and calling setup for those - see the signature of AtomicBasis.setup for explanation.

```python
unique_atoms() -> list[str]
```

Returns list of unique atoms across all molecules

### `basisopt.basis.zetatools` module

```python
basisopt.basis.zetatools.add_np(config: dict[str, int], n: int) -> dict[str, int]
```

Helper function to add n polarization functions

**Parameters**
- **config** – configuration dictionary to polarize
- **n** (*int*) – no. of pol functions to add
BasisOpt

**Returns**
a configuration dictionary

```python
basisopt.basis.zetatools.cc_pv5z(el)
basisopt.basis.zetatools.cc_pvdz(el)
basisopt.basis.zetatools.cc_pvqz(el)
basisopt.basis.zetatools.cc_pvtz(el)
```

```python
basisopt.basis.zetatools.compare(c1: dict[str, int], c2: dict[str, int]) → int
```

Compares two configuration dictionaries

**Returns**

- `< 0` if c1 is bigger than c2
- `0` if they’re equivalent sizes
- `> 0` if c2 is bigger than c1

```python
basisopt.basis.zetatools.config_to_string(conf: dict[str, int]) → str
```

Converts a configuration dictionary into a string, e.g. ‘4s3p2d1f’

```python
basisopt.basis.zetatools.dz(el)
basisopt.basis.zetatools.dzp(el)
basisopt.basis.zetatools.dzpp(el)
```

```python
basisopt.basis.zetatools.enum_shells(conf: ElectronicConfiguration) → dict[str, int]
```

Enumerates the number of functions of each angular momentum

**Parameters**

- `conf` – an ElectronConfig object from mendeleev

**Returns**
a configuration dictionary

```python
basisopt.basis.zetatools.get_next_l(l_list: list[str]) → str
```

Given a list of existing angular momenta, gives the angular momentum symbol one higher.

**Parameters**

- `l_list` (list) – (non-unique) list of angular momenta ['s', 'p', etc]

**Returns**

- angular momentum symbol one higher than max in l_list

```python
basisopt.basis.zetatools.minimal(el)
basisopt.basis.zetatools.n5z(el)
```

```python
basisopt.basis.zetatools.n_cartesian(config: dict[str, int]) → int
```

Returns number of Cartesian Gaussians in configuration

```python
basisopt.basis.zetatools.n_spherical(config: dict[str, int]) → int
```

Returns number of spherical Gaussians in configuration

```python
basisopt.basis.zetatools.nz(el: element, n: int) → dict[str, int]
```

Helper function to generate n-zeta split valence configs

**Parameters**

- `el` (Mendeleev element) –
- `n` (int) – split valence level, e.g. DZ=2, TZ=3, etc.
**Returns**

- a config dictionary

```python
basisopt.basis.zetatools.qz(el)
```

```python
basisopt.basis.zetatools.qzp(el)
```

```python
basisopt.basis.zetatools.qzpp(el)
```

```python
basisopt.basis.zetatools.register_quality(func: Callable[[element], dict[str, int]]) → Callable[[element], dict[str, int]]
```

Decorator to make a quality function available

```python
basisopt.basis.zetatools.string_to_config(string: str) → dict[str, int]
```

converts a configuration string, e.g. '4s3p2d1f', to a Configuration dictionary

```python
basisopt.basis.zetatools.tz(el)
```

```python
basisopt.basis.zetatools.tzp(el)
```

```python
basisopt.basis.zetatools.tzpp(el)
```

**Module contents**

**basisopt.opt package**

**Submodules**

**basisopt.opt.eventemper module**

```python
class basisopt.opt.eventemper.EvenTemperedStrategy(eval_type: str = 'energy', target: float = 1e-05, max_n: int = 18, max_l: int = -1)
```

Bases: `Strategy`

Implements a strategy for an even tempered basis set, where each angular momentum shell is described by three parameters: (c, x, n) Each exponent in that shell is then given by

\[ y_k = c*(x^k) \] for k=0,...,n

**Algorithm:**

- Evaluate: energy (can change to any RMSE-compatible property)
- Loss: root-mean-square error
- Guess: null, uses _INITIAL_GUESS above
- Pre-conditioner: None

**Initialisation:**

- Find minimum no. of shells needed
- max_l >= min_l
- generate initial parameters for each shell

**First run:**

- optimize parameters for each shell once, sequentially

**Next shell in list not marked finished:**

- re-optimise
Uses iteration, limited by two parameters:
max_n: max number of exponents in shell target: threshold for objective function

Additional attributes:
shells (list): list of (c, x, n) parameter tuples shell_done (list): list of flags for whether shell is finished (0) or not (1) target (float): threshold for optimization
delta max_l (int): maximum angular momentum shell to do;

as_dict() → dict[str, Any]
Returns MSONable dictionary of object

classmethod from_dict(d: dict[str, Any]) → object
Creates EvenTemperedStrategy from MSONable dictionary

get_active(basis: dict[str, list[basisopt.containers.Shell]], element: str) → ndarray
Returns the even temper params for the current shell

initialise(basis: dict[str, list[basisopt.containers.Shell]], element: str)
Initialises the strategy by determining the initial parameters for each angular momentum shell for the given element.

next(basis: dict[str, list[basisopt.containers.Shell]], element: str, objective: float) → bool
Moves the strategy forward a step (see algorithm)

set_active(values: ndarray, basis: dict[str, list[basisopt.containers.Shell]], element: str)
Given the even temper params for a shell, expands the basis Checks that the smallest exponent is $\geq 1e^{-5}$ and that the ratio is $\geq 1.01$, to prevent impossible exponents

set_basis_shells(basis: dict[str, list[basisopt.containers.Shell]], element: str)
Expands parameters into a basis set

Parameters

• basis (InternalBasis) – the basis set being optimized
• element (str) – the atom type of interest

Parameters

• basis – internal basis dictionary
• element – symbol of atom being optimized
• objective – value of objective function from last steps

Returns

True if there is a next step, False if strategy is finished

Parameters

• basis (InternalBasis) – the basis set to expand
• element (str) – the atom type
basisopt.opt.optimizers module


General purpose optimizer for a collection of atomic bases

**Arguments:**
- molecules (list): list of Molecule objects to be included in objective basis. Internal basis dictionary, will be used for all molecules.
- opt_data (list): list of tuples, with one tuple for each atomic basis to be optimized. (element, algorithm, strategy, regularizer, opt_params) - see the signature of _atomic_optimize or optimize
- npass (int): number of passes to do, i.e. it will optimize each atomic basis listed in opt_data in order, then loop back and iterate npass times
- parallel (bool): if True, will try to run Molecule calcs in parallel

**Returns**
- dictionary of dictionaries of scipy.optimize results for each step, corresponding to tuple in opt_data

**Raises**
- FailedCalculation


General purpose optimizer for a single atomic basis

**Parameters**
- molecule – Molecule object
- element (str) – symbol of atom to optimize; if None, will default to first atom in molecule
- algorithm (str) – scipy.optimize algorithm to use
- strategy (Strategy) – optimization strategy
- basis_type (str) – which basis type to use; currently “orbital”, “jfit”, or “jkfit”
- reg (func) – regularization function
- opt_params (dict) – parameters to pass to scipy.optimize.minimize

**Returns**
- dictionary of scipy.optimize result objects for each step in the opt

**Raises**
- FailedCalculation
basisopt.opt.preconditioners module

basisopt.opt.preconditioners.inverse\( (\text{inv\_func}: \text{Callable}[[\text{ndarray}, \ldots], \text{ndarray}] \rightarrow \text{Callable}[[\text{ndarray}, \ldots], \text{ndarray}] ) \)

Decorator that adds an inverse function as an attribute All preconditioners must be decorated with an inverse, which should usually have the same signature as the parent.

**Parameters**

- \text{inv\_func (func)} – the inverse of the preconditioner

basisopt.opt.preconditioners.logistic\( \text{(x, minval=0.0001, maxval=100000.0, alpha=1.0, x0=0.0)} \)

Logistic function

basisopt.opt.preconditioners.make_positive\( \text{(x, minval=0.0001, ratio=1.4)} \)

Returns \( \text{x with all values } \geq \text{minval} \) If multiple values are \(< \text{minval}, \text{the new values will be minval } \ast (\text{ratio}**n) \)

basisopt.opt.preconditioners.unit\( \text{(x)} \)

Identity function

basisopt.opt.reduce module


Bases: Strategy

Strategy that takes a basis set and systematically removes least important exponents, until either the change in objective is larger than a threshold value, or a minimal number of exponents is reached.

**Algorithm:**

Evaluate: energy (can change to any RMSE-compatible property) Loss: root-mean-square error Guess: none - initial basis set to reduce must be given Pre-conditioner: any (default, make sure exponents are positive)

Initialization required, to determined parameters for reduction. While delta\_objective is below threshold, and basis size > minimal:

- rank exponents by contribution to objective, for each shell that isn’t already at its minimum size
- remove the least important exponent, adjust basis size
- reoptimize each shell in ascending angular-momentum order
- recalculate delta\_objective

**If delta\_objective > threshold:**

- reset to basis set from previous step

full\_basis

internal basis to be reduced

**Type**

dict
saved_basis
    internal basis from last step
    Type
dict

shells
    list of shells to be reduced
    Type
    list(int)

target
    maximum allowed change in objective value
    Type
    float

method
    method used to evaluate objective
    Type
    str

shell_mins
    minimum number of exponents in each shell, in ascending angular momentum order
    Type
    list(int)

max_l
    maximum angular momentum (inclusive) to reduce
    Type
    int

nexps
    number of exponents in each ang. momentum shell
    Type
    list(int)

reduction_step
    if True, an exponent will be removed when next is called
    Type
    bool

as_dict() \to dict[str, Any]
    Returns MSONable dictionary of object

classmethod from_dict(d: dict[str, Any]) \to object
    Creates ReduceStrategy from MSONable dictionary

 initialise(basis: dict[str, list[basisopt.containers.Shell]], element: str)
    Initialises the strategy by determining the number of exponents in each shell, and making sure we start in
    a reduction step.
next \((basis: dict[str, list[basisopt.containers.Shell]], element: str, objective: float) \rightarrow bool\)

Moves the strategy forward a step (see algorithm)

**Parameters**

- **basis** – internal basis dictionary
- **element** – symbol of atom being optimized
- **objective** – value of objective function from last steps

**Returns**

True if there is a next step, False if strategy is finished

set_basis_shells \((basis: dict[str, list[basisopt.containers.Shell]], element: str)\)

basisopt.opt.regularisers module

basisopt.opt.regularisers.l1_norm \((x: ndarray)\)

basisopt.opt.regularisers.l2_norm \((x: ndarray)\)

basisopt.opt.regularisers.linf_norm \((x: ndarray)\)

basisopt.opt.strategies module

**Class** `basisopt.opt.strategies.Strategy\(eval_type: str = 'energy', pre: ~typing.Callable[[~numpy.ndarray, ...], ~numpy.ndarray] = <function make_positive>\)`

Bases: `MSONable`

Object to describe and handle basis set optimization strategies. All strategy types should inherit from here, and give a description of the approach in the docs. This is a MINIMAL implementation, so all methods here should usually be overridden in child classes

This class also acts as a ‘Default’ optimization strategy. The alg is as follows:

**Algorithm:**

Evaluate: energy (can change to any RMSE-compatible property) Loss: root-mean-square error Guess: cc-pVDZ Pre-conditioner: any (default, make sure exponents are positive)

No initialisation needed. Optimize each shell in increasing order of angular momentum, so self._step = l+1, ends when self._step = max_l+1 No iteration by default.

**name**

identifier

**Type**

str

**eval_type**

property to evaluate

**Type**

str
params
    parameters for backend, see relevant Wrapper for options
    Type
        dict

guess
    function to generate starting guess exponents
    Type
        func
guess_params
    parameters to pass to guess
    Type
        dict
pre
    function to precondition exponents - must have an inverse attribute
    Type
        func
pre_params
    parameters to pass to the preconditioner
    Type
        dict
last_objective
    last value of objective function
    Type
        float
delta_objective
    change in value of objective function from last step
    Type
        float
first_run
    if True, next is yet to be called
    Type
        bool
basis_type
    "orbital/jfit/jkfit", allows for strategy to be applied to auxiliary bases
    Type
        str
orbital_basis
    if using on auxiliary basis, need to specify orbital basis here
    Type
        dict
BasisOpt

loss
function to calculate loss - currently fixed to RMSE

Type
callable

Private attributes:
_step (int): tracks what step of optimization we’re on

as_dict() → dict[str, Any]
Returns MSONable dictionary of Strategy

property eval_type: str

classmethod from_dict(d: dict[str, Any]) → object
Creates a Strategy from MSONable dictionary

get_active(basis: dict[str, list[basisopt.containers.Shell]], element: str) → ndarray

Parameters
• basis – internal basis dictionary
• element – symbol of the atom being optimized

Returns
the set of exponents currently being optimised

initialise(basis: dict[str, list[basisopt.containers.Shell]], element: str)
Initialises the strategy (does nothing in default)

Parameters
• basis – internal basis dictionary
• element – symbol of the atom being optimized

next(basis: dict[str, list[basisopt.containers.Shell]], element: str, objective: float) → bool
Moves the strategy forward a step (see algorithm)

Parameters
• basis – internal basis dictionary
• element – symbol of atom being optimized
• objective – value of objective function from last steps

Returns
True if there is a next step, False if strategy is finished

set_active(values: ndarray, basis: dict[str, list[basisopt.containers.Shell]], element: str)
Sets the currently active exponents to the given values.

Parameters
• values (list) – list of new exponents
• basis – internal basis dictionary
• element – symbol of atom being optimized
Module contents

basisopt.testing package

Submodules

basisopt.testing.dunham module

class basisopt.testing.dunham.DunhamTest(name: str, mol: Molecule | None = None, mol_str: str = '', charge: int = 0, mult: int = 1, poly_order: int = 6, step: float = 0.05, Emax: float = 0)

Bases: Test

Carries out a Dunham analysis on a diatomic, calculating spectroscopic constants. Initialised with either a diatomic Molecule object, or a mol_string of the form “Atom1Atom2, separation in Ang”, e.g. “H2,0.9”, “NO,1.2”, “LiH,1.3” etc.

Results:

- returned as numpy array, as well as archived Ee: Energy at eq. separation (Ha)
- Re: Eq. separation (Ang)
- BRot, ARot: First and second rotational constants (cm-1)
- We: First vibrational constant (cm-1)
- Wx, Wy: x and y anharmonic corrections to We (cm-1)
- De: Dissociation energy (eV)
- D0: Zero-point dissociation energy (eV)

Additional data stored:

- StencilRi (numpy array): the separation values (Ang) used in the polynomial fit
- StencilEi (numpy array): the energy values (Ha) at each point in the fit

Additional attributes:

- poly_order (int): order of polynomial to fit, >= 3
- step (float): step size in Angstrom to use for polynomial fit
- Emax (float): energy in Ha to calculate dissociation from (default 0)
- poly (poly1d): fitted polynomial
- shift (float): the shift for separations used in the polynomial fit, e.g. to calculate the value at the point R, use poly(R-shift)

as_dict()  
Converts Result (and all children) to an MSONable dictionary


Calculates the Dunham analysis results

Parameters

- method (str): calculation method to use
- basis (InternalBasis): orbital basis set
- params (dict): parameters to pass to backend

Returns

array of results in order specified by _VALUE_NAMES

classmethod from_dict(d)

Creates Result from dictionary representation, including recursive creation of children.

from_string(mol_str: str, charge: int = 0, mult: int = 1)

Makes a diatomic molecule from string to use in test

Parameters
BasisOpt

- **mol_str** (str) – string of diatomic and separation in Angstrom e.g. “NO,1.3”, “H2,0.9”, “LiH,1.1” etc
- **charge** (int) – overall charge of diatomic
- **mult** (int) – spin multiplicity of diatomic

**reduced_mass() → float**

Calculate the reduced mass of the diatomic

```python
basisopt.testing.dunham.dunham(energies: ndarray, distances: ndarray, mu: float, poly_order: int = 6, angstrom: bool = True, Emax: float = 0) → tuple[numpy.poly1d, float, numpy.ndarray]
```

Performs a Dunham analysis on a diatomic, given energy/distance values around a minimum and the reduced mass mu

### basisopt.testing.rank module

**basisopt.testing.rank.rank_primitives**

```python
basisopt.testing.rank.rank_primitives(atomic: AtomicBasis, shells: list[int] | None = None, eval_type: str = 'energy', basis_type: str = 'orbital', params={})
```

Systematically eliminates exponents from shells in an AtomicBasis to determine how much they contribute to the target property

**Parameters**

- **atomic** – AtomicBasis object
- **shells** (list) – list of indices for shells in the AtomicBasis to be ranked. If None, will rank all shells
- **eval_type** (str) – property to evaluate (e.g. energy)
- **basis_type** (str) – “orbital/jfit/jkfit”
- **params** (dict) – parameters to pass to the backend, see relevant Wrapper for options

**Returns**

tuples (errors, ranks), where errors is a list of numpy arrays with the change in target property value for each exponent in the shell, and ranks is a list of numpy arrays which contain the indices of each exponent in each shell from smallest to largest error value. Order of errors, ranks is same as order of shells

**Raises**

- **FailedCalculation** –

**basisopt.testing.rank.reduce_primitives**

```python
basisopt.testing.rank.reduce_primitives(atomic: AtomicBasis, thresh: float = 0.0001, shells: list[int] | None = None, eval_type: str = 'energy', params: dict[str, Any] = {}) → tuple[dict[str, list[basisopt.containers.Shell]], Any]
```

Rank the primitive functions in an atomic basis, and remove those that contribute less than a threshold. TODO: add checking that does not go below minimal config

**Parameters**

- **atomic** – AtomicBasis object
- **thresh** (float) – if a primitive’s contribution to the target is < thresh,
- **basis** (it is removed from the) –
- **shells** (list) – list of indices of shells to be pruned; if None, does all shells

24 Chapter 2. Citation
- **eval_type** (*str*) – property to evaluate
- **params** (*dict*) – parameters to pass to the backend

**Returns**

(basis, delta) where basis is the pruned basis set (this is non-destructive to the original AtomicBasis), and delta is the change in target property with the pruned basis compared to the original

**Raises**

`FailedCalculation` –

---

**basisopt.testing.test module**

**class** `basisopt.testing.test.PropertyTest`(name: *str*, prop: *str* = 'energy', mol: `Molecule` | *None* = *None*, xyz_file: *str* | *None* = *None*, charge: *int* = 0, mult: *int* = 1)

**Bases:** `Test`

Simplest implementation of Test, calculating some property, e.g. energy

**Additional attributes:**

- `eval_type` (*str*): property to evaluate, e.g. ‘energy’, ‘dipole’

**as_dict()**

Converts Result (and all children) to an MSONable dictionary


Calculates the test value

**Parameters**

- **method** (*str*) – the method to use, e.g. ‘scf’
- **basis** (*dict*) – internal basis object
- **params** (*dict*) – parameters to pass to backend

**Returns**

the value from the calculation

**property eval_type:** *str*

**classmethod** `from_dict(d)`

Creates Result from dictionary representation, including recursive creation of children.

**class** `basisopt.testing.test.Test`(name: *str*, reference: `Any` | *None* = *None*, mol: `Molecule` | *None* = *None*, xyz_file: *str* | *None* = *None*, charge: *int* = 0, mult: *int* = 1)

**Bases:** `Result`

Abstract Test class, a type of Result, for a way of testing a basis set, see e.g. PropertyTest and DunhamTest

**reference**

reference value of some kind to compare test result to

**Type**

var

**molecule**

Molecule object to perform test with

**Must implement in children:**

`calculate(self, method, basis, params={})`
BasisOpt

as_dict()
Converts Result (and all children) to an MSONable dictionary

Interface to run the test. Should archive and return the results of the test.

method
method to run, e.g. ‘rhf’, ‘mp2’
Type str

basis
internal basis dictionary

params
parameters to pass to the backend Wrapper
Type dict

Calculates reference value for the test, should not need to be overridden

method
method to run, e.g. ‘rhf’, ‘mp2’
Type str

basis
internal basis dictionary

basis_name
calculate using basis with this name from BSE, if basis is None
Type str

params
parameters to pass to the backend Wrapper
Type dict

classmethod from_dict(d)
Creates Result from dictionary representation, including recursive creation of children.

set_molecule_from_xyz(xyz: str, charge: int = 0, mult: int = 1)
Creates Molecule from xyz file

Parameters
• xyz (str) – the xyz file
• charge (int), mult (int) – the charge and multiplicity of the molecule
Module contents

basisopt.wrappers package

Submodules

basisopt.wrappers.dummy module

class basisopt.wrappers.dummy.DummyWrapper
    Bases: Wrapper
    A Wrapper that can’t actually be used to do computations. It has two purposes: to be a default Wrapper so that the library can be used even when calculations aren’t needed (e.g. if looking at and analysing previously computed results); and to make testing a lot easier, as we don’t have to have e.g. Psi4 installed. As such this has minimal functionality.

    convert_molecule(m: Molecule) → int
        Dummy molecule converter

    dipole(mol, tmp='')
        Dipole moment, numpy array, a.u.

    energy(mol, tmp='')
        Energy, Hartree

    initialise(m: Molecule, name: str = '', tmp: str = '')
        Initialises calc by converting molecule, setting globals and self._basis_value

    polarizability(mol, tmp='')
        Dipole polarizability, a.u.

    quadrupole(mol, tmp='')
        Quadrupole moment, numpy array, a.u.

basisopt.wrappers.orca module

class basisopt.wrappers.orca.OrcaWrapper(orca_path: str)
    Bases: Wrapper
    Wrapper for Orca 5

    Private attribute:
        pwd (str): the present working directory

    convert_molecule(m: Molecule) → str
        Convert an internal Molecule object to an Orca geometry section

    dipole(mol, tmp='', **params)
        Dipole moment, numpy array, a.u.

    energy(mol, tmp='', **params)
        Energy, Hartree

2.4. BasisOpt API
**initialise**

```python
initialise(m: Molecule, name: str = '', tmp: str = '.', **params) → str
```

Initialises Orca and creates input file for calculation. WARNING: moves to the scratch directory, but saves PWD in

```python
self._pwd
```

**Parameters**

- `m (Molecule)` – molecule to run calculation on
- `name (str)` – name for calculation
- `tmp (str)` – path to scratch directory

**Returns**

the prefix for the calculation input/output files

**jk_error**

```python
jk_error(mol, tmp='', **params)
```

JK density fitting error, Hartree

**polarizability**

```python
polarizability(mol, tmp='', **params)
```

Dipole polarizability, a.u.

**quadrupole**

```python
quadrupole(mol, tmp='', **params)
```

Quadrupole moment, numpy array, a.u.

---

**basisopt.wrappers.psi4 module**

**basisopt.wrappers.wrapper module**

```python
class basisopt.wrappers.wrapper_WRAPPER(name: str = 'Empty')
```

Abstract class to derive all backend wrappers from; see e.g. Psi4.

All new calculation types must be added to this class and marked as unavailable, this means that the library knows what calculation types are possible agnostic to which wrapper is being used.

You only have to implement the calculation types that you want to expose, and decorate them as being available. These functions should have the signature

```python
func(self, mol, tmp='')
```

where mol is a Molecule object, and tmp is the path to the scratch directory.

**Attributes that should be set in children:**

- `_name (str)`: identifier, e.g. 'Psi4'
- `_method_strings (dict)`: dictionary of method names and what calculation types can be done with them, e.g. {'hf': ['energy', 'dipole'], 'mp2': ['energy'], ...}

**Attributes used by children:**

- `_values (dict)`: dictionary where most recent calculated results are stored
- `_globals (dict)`: dictionary of parameters that should be set every time a calculation is run, e.g. {'memory': '2gb', ...}. These should be parsed as part of the ‘run’ function in every Child implementation

**Attributes that should only be set here:**

- `_methods (dict)`: dictionary of all possible calculation types, pointing to member funcs

```python
add_global(name: str, value: Any)
```

Add a global option
all_available() → list[str]
Returns a list of all available calculation types

available_methods(prop: str) → list[str]
Returns a list of all available methods to calculate a particular property
prop
  name of property, e.g. ‘energy’, ‘dipole’
  Type
  str

available_properties(name: str) → list[str]
Returns a list of all available calculation types for a given method.
name
  method name, e.g. ‘rhf’, ‘mp2’
  Type
  str

clean()
Cleans up any temporary files
dipole(mol, tmp='', **params)
  Dipole moment, numpy array, a.u.
energy(mol, tmp='', **params)
  Energy, Hartree
get_value(name: str) → Any
  Retrieve a data point if it exists
jk_error(mol, tmp='', **params)
  JK density fitting error, Hartree
method_is_available(method: str = 'energy') → bool
  Returns True if a calculation type is available, false otherwise
polarizability(mol, tmp='', **params)
  Dipole polarizability, a.u.
quadrupole(mol, tmp='', **params)
  Quadrupole moment, numpy array, a.u.
run(evaluate: str, molecule: Molecule, params: dict[str, Any], tmp: str = '') → int
  Runs a calculation with this backend MUST BE IMPLEMENTED IN ALL WRAPPERS
Parameters
  • evaluate (str) – the property to evaluate, e.g. ‘energy’
  • molecule – a Molecule object to run a calculation on
  • params (dict) – any parameters for the calculation in addition to _globals
  • tmp (str) – path to scratch directory
Returns
  0 on success, -1 if method isn’t available, -2 otherwise
**trans_dipole**(mol, tmp='') **params**
Transition dipole moment, numpy array, a.u.

**trans_quadrupole**(mol, tmp='', **params**)
Transition quadrupole moment, numpy array, a.u.

**verify_method_string**(string: str) → bool
Checks whether a method is available with this wrapper

**Parameters**

- **string** (str) – a string of the form “name.method”, e.g. “rhf.energy”
- **'rhf'** (will check to see if 'energy' can be calculated with)

**Returns**

True if available, False otherwise

**Raises**

InvalidMethodString

basisopt.wrappers.wrapper.available(func: Callable[[object, Molecule, str, ...], Any]) → Callable[[object, Molecule, str, ...], Any]
Decorator to mark a method as available

basisopt.wrappers.wrapper.unavailable(func: Callable[[object, Molecule, str, ...], Any]) →
Callable[[object, Molecule, str, ...], Any]
Decorator to mark a method as unavailable

**Module contents**

**Submodules**

**basisopt.api module**

basisopt.api.dummy(path: str)
Sets backend to the DummyWrapper for testing and for when calculations aren’t needed.

basisopt.api.get_backend() → Wrapper
Returns: backend (Wrapper): the Wrapper object for the current backend

basisopt.api.get_tmp_dir() → str
Returns: Path to the current scratch/temp directory

basisopt.api.orca(path: str)
Tests orca import and prepares to be used as calculation backend

basisopt.api.psi4(path: str)
Tests Psi4 import and prepares to be used as calculation backend

basisopt.api.register_backend(func: Callable[[str, str], None]) → Callable[[str, str], None]
Registers a function to set the backend for basisopt

Runs calculations over a set of molecules, optionally in parallel

**Parameters**
### BasisOpt

- **evaluate**(str) – the property to evaluate
- **mols**(list) – a list of Molecule objects to run
- **params**(dict) – parameters for backend
- **parallel**(bool) – if True, will try to run distributed

**Returns**

- a dictionary of the form {molecule name: value}

**Return type**

- a dictionary of the form {molecule name: value}

```python
basisopt.api.run_calculation(evaluate: str = 'energy', mol: Molecule | None = None, params: dict[Any, Any] = {})
```

Interface to the wrapper used to run a calculation.

**Parameters**

- **evaluate**(str) – The function to be called for the computation
- **mol**(Molecule) – molecule to run the calculation on
- **params**(dict) – A dictionary of parameters needed for the computation

**Returns**

- 0 on success, non-zero on failure

**Return type**

- int

```python
basisopt.api.set_backend(name: str, path: str = '')
```

Sets the global backend for basisopt calculations

**Parameters**

- **name**(str) – the name of the program to use
- **path**(str) – absolute path to the program executable

```python
basisopt.api.set_logger(level: int = 20, filename: str | None = None)
```

Initialises Python logging, formatting it nicely, and optionally printing to a file.

```python
basisopt.api.set_parallel(value: bool = True)
```

Turns parallelism on/off

```python
basisopt.api.set_tmp_dir(path: str)
```

Sets the working directory for all backend calculations, creating the directory if it doesn’t already exist.

**Parameters**

- **path**(str) – path to the scratch directory

```python
basisopt.api.which_backend() -> str
```

Returns: str: The name of the currently registered backend
basisopt.bse_wrapper module

basisopt.bse_wrapper.bse_to_internal(basis: dict[str, Any]) → dict[str, list[basisopt.containers.Shell]]

Converts a BSE basis object into an internal basis dictionary

Parameters

basis – a BSE basis, must have the following attributes ['elements'] each of which must then have an ['electron_shells'] attribute

Returns

an internal basis dictionary

basisopt.bse_wrapper.fetch_basis(name: str, elements: list[str]) → dict[str, list[basisopt.containers.Shell]]

Fetches a basis set for a set of elements from the BSE

Parameters

• name (str) –
• elements (list) - a list of element symbols (or atomic numbers)

Returns

an internal basis dictionary

basisopt.bse_wrapper.fetch_ecp(name: str, elements: list[str]) → dict[str, Any]

Fetches an ECP basis from the BSE, deleting any extraneous non-ECP info

Parameters

• name – name of desired basis
• elements – list of element symbols or atomic numbers

Returns

a BSE basis dictionary

basisopt.bse_wrapper.internal_basis_converter(basis: dict[str, list[basisopt.containers.Shell]], fmt: str = 'gaussian94') → str

Writes out an internal basis in the desired BSE format

Parameters

• basis (dict) – the internal basis dictionary
• fmt (str) – the desired output format - see the BSE docs for options

Returns

the basis as a string in the desired format

basisopt.bse_wrapper.internal_to_bse(basis: dict[str, list[basisopt.containers.Shell]]) → dict[str, Any]

Converts an internal basis dictionary into a BSE basis object

Parameters

basis – an internal basis, which is a dictionary with k, v pairs like: element_symbol: [array of internal Shell objects]

Returns

a BSE basis of type 'component' with 'gto_spherical' function types

basisopt.bse_wrapper.make_bse_shell(shell: Shell) → dict[str, Any]

Converts an internal-format basis shell into a BSE-format shell

Parameters

shell – an internal Shell object
Returns
a BSE-format gto_spherical shell

\texttt{basisopt.bse\_wrapper.make\_internal\_shell}(\texttt{shell: dict[str, Any]}) \rightarrow \texttt{Shell}
Converts a BSE-format basis shell into an internal-format shell

Parameters
\begin{itemize}
    \item \texttt{shell} -- a BSE shell, a dictionary that must have these attributes ['angular\_momentum', 'exponents', 'coefficients']
\end{itemize}

Returns
an internal Shell object

\textbf{basisopt.containers module}

class \texttt{basisopt.containers.Result}(\texttt{name: str = 'Empty'})
Container for storing and archiving all results, e.g. of tests, calculations, and optimizations.

\begin{itemize}
    \item \texttt{name} -- identifier for result
    \item \texttt{depth} -- a Result object contains children, so a depth of 1 indicates no parents, 2 indicates one parent, etc.
\end{itemize}

Private attributes:
\begin{itemize}
    \item \texttt{_data\_keys (dict): dictionary with the format}
        (value\_name, number of records)
    \item \texttt{_data\_values (dict): dictionary of values with format}
        (value\_name\_with\_id, value)
    \item \texttt{_children (list): references to child Result objects}
\end{itemize}

\texttt{add\_child}(\texttt{child: object})
Adds a child Result to this Result

\texttt{add\_data}(\texttt{name: str, value: Any})
Adds a data point to the result, with archiving

Parameters
\begin{itemize}
    \item \texttt{name} (\texttt{str}) -- identifier for the value
    \item \texttt{value} -- the value, can be basically anything
\end{itemize}

\texttt{as\_dict()} \rightarrow \texttt{dict[str, Any]}
Converts Result (and all children) to an MSONable dictionary

property \texttt{depth: int}
classmethod `from_dict(d: dict[str, Any]) → object`

Creates Result from dictionary representation, including recursive creation of children.

`get_child(name: str) → object`

Returns child Result with given name, if it exists

`get_data(name: str, step_back: int = 0) → Any`

Retrieve an archived data point

Parameters

- `name (str)` – identifier for the value needed
- `step_back (int)` – how many values back to go,
- `added (default will return last point)` –

Returns

the value with the requested name, if it exists

Raises

`DataNotFound if the requested data doesn't exist` –

`load(filename: str) → object`

Loads and returns a Result object from a file pickle

`save(filename: str)`

Pickles the Result object into a file

`search(name: str) → dict[str, Any]`

Searches for all data in this and all its children with a given name, returning a dictionary indexed by the name and which child it was found in

`statistics()`

Tabulates summary statistics for the data in this Result Note: does not recur over children

`summary() → str`

Creates summaries of the Result and all its children

Returns

a string with human-readable summary of the results

class `basisopt.containers.Shell`

Bases: `MSONable`

Lightweight container for basis set Shells.

1

the angular momentum name of the shell

Type

`char`

`exps`

array of exponents

Type

`numpy array, float`
**coefs**

list of numpy arrays of equal length to exps, corresponding to coefficients for each exponent

**Type**

list

**as_dict() → dict[str, Any]**

Converts Shell to MSONable dictionary

**Returns**

dictionary representation of Shell

**compute(x: float, y: float, z: float, i: int = 0, m: int = 0) → float**

Computes the value of the (spherical) GTO at a given point

**Parameters**

- **x (float)** – coordinates relative to center of GTO
- **y (float)** – coordinates relative to center of GTO
- **z (float)** – coordinates relative to center of GTO
- **i (int)** – index of GTO in coefs
- **m (int)** – azimuthal quantum number in [-l, l]

**Returns**

The unnormalised value of the GTO at (x, y, z)

**classmethod from_dict(d: dict[str, Any]) → object**

Creates Shell object from dictionary representation

**Parameters**

d (dict) – dictionary of Shell attributes

**Returns**

Shell object

**basisopt.containers.basis_to_dict(basis: dict[str, list[basisopt.containers.Shell]]) → dict[str, Any]**

Converts an internal basis set of the form {atom: [shells]} to an MSONable dictionary

**Parameters**

- **basis (dict)** – internal basis set

**Returns**

json-writable dictionary

**basisopt.containers.dict_to_basis(d: dict[str, Any]) → dict[str, list[basisopt.containers.Shell]]**

Converts an MSON dictionary to an internal basis

**Parameters**

- **d (dict)** – dictionary of basis set attributes

**Returns**

internal basis set
**basisopt.data module**

```python
basisopt.data.AM_DICT = {'d': 2, 'f': 3, 'g': 4, 'h': 5, 'i': 6, 'j': 7, 'k': 8, 'l': 9, 'p': 1, 's': 0}

Dictionary converting back from l quantum number to letter value
```

```python

Dictionary with pre-optimised even-tempered expansions for atoms
```

```python
basisopt.data.atomic_number(element: str) → int

basisopt.data.get_even_temper_params(atom: str = 'H', accuracy: float = 1e-05) → list[tuple[float, float, int]]

Searches for the relevant even tempered expansion from _EVEN_TEMPERED_DATA
```

**basisopt.exceptions module**

```python
exception basisopt.exceptions.DataNotFound
    Bases: Exception

exception basisopt.exceptions.ElementNotSet
    Bases: Exception

exception basisopt.exceptions.EmptyBasis
    Bases: Exception

exception basisopt.exceptions.EmptyCalculation
    Bases: Exception

exception basisopt.exceptions.FailedCalculation
    Bases: Exception

exception basisopt.exceptions.InvalidDiatomic
    Bases: Exception

exception basisopt.exceptions.InvalidMethodString
    Bases: Exception

exception basisopt.exceptions.InvalidResult
    Bases: Exception

exception basisopt.exceptions.MethodNotAvailable(estr)
    Bases: Exception

exception basisopt.exceptions.PropertyNotAvailable(pstr)
    Bases: Exception
```
class basisopt.molecule.Molecule

Bases: MSONable

A very loose definition of a molecule, in that it represents an object with which calculations can be done.

**name**

identifier

    Type
    str

**charge**

overall net charge

    Type
    int

**multiplicity**

spin multiplicity, i.e. 2S+1

    Type
    int

**method**

name of calculation method, e.g. ‘hf’ or ‘ccsd(t)’

    Type
    str

**basis**

internal basis dictionary, which has (k, v) pairs

    Type
    dict

**ecps**

map of atom types to ecp basis names

    Type
    dict

**jbasis**

internal basis dictionary for Coulomb fitting set

    Type
    dict

**jkbasis**

internal basis dictionary for Coulomb+Exchange fitting set

    Type
    dict

_of the form_ (element_symbol

array of Shell objects)
**dummy_atoms**

list of indices of atoms that should be treated as dummies

**Type**

list

**Private attributes:**

Atom symbols in order, e.g. ['H', 'H', 'O'] _coords (list): x,y,z coords in Angstrom, as numpy arrays, same order as _atom_names _results (dict): dictionary of results calculated for this molecule.

NOTE: these results are NOT archived, unlike for a Result object _references (dict): dictionary of reference values for results

**add_atom**(element: str = 'H', coord: list[float] = [0.0, 0.0, 0.0], dummy: bool = False)

Adds an atom to the molecule

**Parameters**

- element (str) – element name
- coord (list) – [x,y,z] coords in Angstrom
- dummy (bool) – if True, the atom is marked as a dummy atom

**add_reference**(name: str, value: Any)

Same as add_result but for reference values

**add_result**(name: str, value: Any)

Store a result (no archiving)

**Parameters**

- name (str) – identifier for result
- value (any) – value of result

**as_dict**() → dict[str, Any]

Converts Molecule to MSONable dictionary

**Returns**

dictionary representing the molecule

**distance**(atom1: int, atom2: int) → float

Computes the Euclidean distance between two atoms. No bounds checking.

**Parameters**

- atom1 (int) – indices of atoms
- atom2 (int) – indices of atoms

**Returns**
	he Euclidean separation in Angstrom

**classmethod from_dict**(d: dict[str, Any]) → object

Creates a Molecule from a dictionary

**Parameters**

d (dict) – dictionary with Molecule attributes

**Returns**

Molecule
classmethod from_xyz(filename: str, name: str = 'Untitled', charge: int = 0, mult: int = 1) → object

    Creates a Molecule from an xyz file

    Parameters
    filename (str) – path to xyz file

get_delta(name: str) → Any

    Returns: Difference between a result and its reference value

get_line(i: int, atom_prefix: str = '', atom_suffix: str = '') → str

    Gets a line of the xyz file representation of the Molecule

    Parameters
    • i (int) – the index of the atom line wanted
    • atom_prefix (str) – optional string to add at start of atom name (for e.g. dummy atoms in psi4)
    • atom_suffix (str) – optional string to add at end of atom name (for e.g. dummy atoms in Orca)

    Returns
    a string of form {prefix+element+suffix} {coords}

get_reference(name: str) → Any

    Same as get_result but for reference values

get_result(name: str) → Any

    Returns: Value of result with given name if it exists, otherwise 0

natoms() → int

    Returns number of atoms in Molecule

nelectrons() → int

    Returns the number of electrons in the molecule, not accounting for any ECPs

set_dummy_atoms(indices: list[int], overwrite: bool = True)

    Sets the list of atoms that should be considered dummies or ghosts

    Parameters
    • indices – list of indices specifying which atoms to dummy-ify
    • overwrite – if True, will overwrite any existing list of dummies, otherwise will append to the existing list

set_ecps(ecp_dict: dict[str, str])

    Sets the ECP dictionary.

    Parameters
    ecp_dict – a dictionary of atom name to ECP name. The ECP name should be either a name from BSE (for Psi4 backend), or the Orca internal library (for Orca backend, list of names can be found in the manual)

to_xyz() → str

    Converts Molecule to xyz file format

    Returns
    a string of the Molecule in xyz file format
unique_atoms() → list[str]
Returns a list of all unique atom types in Molecule

basisopt.molecule.build_diatomic(mol_str: str, charge: int = 0, mult: int = 1) → Molecule
Builds a diatomic molecule from a string

Parameters

- mol_str (str) – string of diatomic and separation in Angstrom
- "NO (e.g.) –
- 1.3 –
- "H2 –
- 0.9 –
- "LiH –
- etc (1.1") –
- charge (int) – net molecular charge
- mult (int) – spin multiplicity

Returns
Molecule object of diatomic

Raises

- IndexError when rval not given in mol_str –
- InvalidDiatomic when mol_str can't be parsed –
- error checking not exhaustive –

basisopt.parallelise module

basisopt.parallelise.chunk(x: list[Any], n_chunks: int) → list[list[Any]]
Chunks an array into roughly equal-sized subarrays

Parameters

- L (x - array of values of length) –
- into (n_chunks - number of chunks to split) –

Returns
a list of n_chunks arrays of length L//n_chunks or (L//n_chunks)+1

basisopt.parallelise.distribute(n_proc: int, func: Callable[[list[Any], dict], Any], x: list[Any], **kwargs) → list[Any]
Distributes a function over a desired no. of procs using the distributed library.

Parameters

- start (n_proc - the number of processes to) –
- call (func - the function to) –
- signature (with) –
- over (x - the array of values to distribute) –
- func (kwargs - the named arguments accepted by) –
Returns
a list of results ordered by process ID

**basisopt.util module**

`basisopt.util.dict_decode(d: dict[str, Any]) → dict[str, Any]`

`basisopt.util.fit_poly(x: ndarray, y: ndarray, n: int = 6) → tuple[numpy.poly1d, float, float, list[float]]`

Fits a polynomial of order \( n \) to the set of \((x \text{ [Bohr]}, y \text{ [Hartree]})\) coordinates given, and calculates data necessary for a Dunham analysis.

**Parameters**

- \( x \text{ (numpy array)} \) – atomic separations in Bohr
- \( y \text{ (numpy array)} \) – energies at each point in Hartree
- \( n \text{ (int)} \) – order of polynomial to fit

**Returns**

poly1d object, reference separation (Bohr), equilibrium separation (Bohr), first \((n+1)\) Taylor series coefficients at eq. sep.

`basisopt.util.read_json(filename: str) → MSONable`

Reads an MSONable object from file

**Parameters**

- \( filename \text{ (str)} \) – path to JSON file

**Returns**

object

`basisopt.util.write_json(filename: str, obj: MSONable)`

Writes an MSONable object to file

**Parameters**

- \( filename \text{ (str)} \) – path to JSON file
- \( obj \text{ (MSONable)} \) – object to be written

**basisopt.version module**

**Module contents**
CHAPTER THREE

INDICES AND TABLES

- genindex
- modindex
- search
basisopt, 41
basisopt.api, 30
basisopt.basis, 15
basisopt.basis.atomic, 7
basisopt.basis.basis, 10
basisopt.basis.guesses, 11
basisopt.basis.molecular, 12
basisopt.basis.zetatools, 13
basisopt.bse_wrapper, 32
basisopt.containers, 33
basisopt.data, 36
basisopt.exceptions, 36
basisopt.molecule, 37
basisopt.opt, 23
basisopt.opt.eventemper, 15
basisopt.opt.optimizers, 17
basisopt.opt.preconditioners, 18
basisopt.opt.reduce, 18
basisopt.opt.regularisers, 20
basisopt.opt.strategies, 20
basisopt.parallelise, 40
basisopt.testing, 27
basisopt.testing.dunham, 23
basisopt.testing.rank, 24
basisopt.testing.test, 25
basisopt.util, 41
basisopt.wrappers, 30
basisopt.wrappers.dummy, 27
basisopt.wrappers.orca, 27
basisopt.wrappers.wrapper, 28
INDEX

A
add_atom() (basisopt.molecule.Molecule method), 38
add_child() (basisopt.containers.Result method), 33
add_data() (basisopt.containers.Result method), 33
add_global() (basisopt.wrappers.wrapper.Wrapper method), 28
add_molecule() (basisopt.basis.molecular.MolecularBasis method), 12
add_np() (in module basisopt.basis.zetatools), 13
add_reference() (basisopt.molecule.Molecule method), 38
add_result() (basisopt.molecule.Molecule method), 38
all_available() (basisopt.wrappers.wrapper.Wrapper method), 28
AM_DICT (in module basisopt.data), 36
as_dict() (basisopt.basis.atomic.AtomicBasis method), 8
as_dict() (basisopt.basis.basis.Basis method), 10
as_dict() (basisopt.basis.molecular.MolecularBasis method), 12
as_dict() (basisopt.containers.Result method), 33
as_dict() (basisopt.containers.Shell method), 35
as_dict() (basisopt.molecule.Molecule method), 38
as_dict() (basisopt.opt.eventemper.EvenTemperedStrategy method), 16
as_dict() (basisopt.opt.reduce.ReduceStrategy method), 19
as_dict() (basisopt.opt.strategies.Strategy method), 22
as_dict() (basisopt.testing.dunham.DunhamTest method), 23
as_dict() (basisopt.testing.test.PropertyTest method), 25
as_dict() (basisopt.testing.test.Test method), 26
as_xyz() (basisopt.basis.atomic.AtomicBasis method), 8
atomic_number() (in module basisopt.data), 36
AtomicBasis (class in basisopt.basis.atomic), 7
available() (in module basisopt.wrappers.wrapper), 30
available_methods() (basisopt.wrappers.wrapper.Wrapper method), 29
available_properties() (basisopt.wrappers.wrapper.Wrapper method), 29

B
basis (basisopt.basis.molecular.MolecularBasis attribute), 12
basis (basisopt.molecule.Molecule attribute), 37
basis (basisopt.testing.test.Test attribute), 26
Basis (class in basisopt.basis.basis), 10
basis_name (basisopt.testing.test.Test attribute), 26
basis_to_dict() (in module basisopt.containers), 35
basis_type (basisopt.strategies.Strategy attribute), 21
basisopt module, 41
basisopt.api module, 30
basisopt.basis module, 15
basisopt.basis.atomic module, 7
basisopt.basis.basis module, 10
basisopt.basis.guesses module, 11
basisopt.basis.molecular module, 12
basisopt.basis.zetatools module, 13
basisopt.bse_wrapper module, 32
basisopt.containers module, 33
basisopt.data module, 36
basisopt.exceptions module, 36
basisopt.molecule module, 37
basisopt.opt module, 23
basisopt.opt.eventemper
INDEX

1 (basisopt.containers.Shell attribute), 34
11_norm() (in module basisopt.opt.regularisers), 20
12_norm() (in module basisopt.opt.regularisers), 20
last_objective (basisopt.opt.strategies.Strategy attribute), 21
linf_norm() (in module basisopt.opt.regularisers), 20
load() (basisopt.basis.basis.Basis method), 10
load() (basisopt.containers.Result method), 34
log_normal_guess() (in module basisopt.basis.guesses), 11
logistic() (in module basisopt.opt.preconditioners), 18
loss (basisopt.opt.strategies.Strategy attribute), 21

M
make_bse_shell() (in module basisopt.bse_wrapper), 32
make_internal_shell() (in module basisopt.bse_wrapper), 33
make_positive() (in module basisopt.opt.preconditioners), 18
max_l (basisopt.opt.reduce.ReduceStrategy attribute), 19
method (basisopt.basis.Molecule attribute), 37
method (basisopt.opt.reduce.ReduceStrategy attribute), 19
method (basisopt.testing.test.Test attribute), 26
method_is_available() (basisopt.wrappers.wrapper.Wrapper method), 29
MethodNotAvailable, 36
minimal() (basisopt.basis.atomic.AtomicBasis method), 8
minimal() (in module basisopt.basis.zetasools), 14
module
basisopt, 41
basisopt.api, 30
basisopt.basis, 15
basisopt.basis.atomic, 7
basisopt.basis.basis, 10
basisopt.basis.guesses, 11
basisopt.basis.molecular, 12
basisopt.basis.zetasools, 13
basisopt.bse_wrapper, 32
basisopt.containers, 33
basisopt.data, 36
basisopt.exceptions, 36
basisopt.molecule, 37
basisopt.opt, 23
basisopt.opt.eventemper, 15
basisopt.opt.optimizers, 17
basisopt.opt.preconditioners, 18
basisopt.opt.reduce, 18
basisopt.opt.regularisers, 20
basisopt.opt.strategies, 20
basisopt.parallelise, 40
basisopt.testing, 27
basisopt.testing.dunham, 23
basisopt.testing.rank, 24
basisopt.testing.test, 25
basisopt.util, 41
basisopt.wrappers, 30
basisopt.wrappers.dummy, 27
basisopt.wrappers.orca, 27
basisopt.wrappers.wrapper, 28

MolecularBasis (class in basisopt.basis.molecular), 12
molecule (basisopt.testing.test.Test attribute), 25
Molecule (class in basisopt.molecule), 37
molecules() (basisopt.basis.molecular.MolecularBasis method), 12
multiplicity (basisopt.basis.atomic.AtomicBasis attribute), 8
multiplicity (basisopt.basis.atomic.AtomicBasis property), 8
multiplicity (basisopt.molecule.Molecule attribute), 37

N
n5z() (in module basisopt.basis.zetasools), 14
n_cartesian() (in module basisopt.basis.zetasools), 14
n_spherical() (in module basisopt.basis.zetasools), 14
name (basisopt.containers.Result attribute), 33
name (basisopt.molecule.Molecule attribute), 37
name (basisopt.opt.strategies.Strategy attribute), 20
name (basisopt.wrappers.wrapper.Wrapper attribute), 29
natom() (basisopt.molecule.Molecule method), 39
needs_element() (in module basisopt.basis.atomic), 10
nelectrons() (basisopt.molecule.Molecule method), 39
nexps (basisopt.opt.reduce.ReduceStrategy attribute), 19
next() (basisopt.opt.eventemper.EventTemperedStrategy method), 16
next() (basisopt.opt.reduce.ReduceStrategy method), 19
next() (basisopt.opt.strategies.Strategy method), 22
null_guess() (in module basisopt.basis.guesses), 12
nz() (in module basisopt.basis.zetatools), 14

O
optimize() (basisopt.basis.atomic.AtomicBasis method), 9
optimize() (basisopt.basis.basis.Basis method), 10
optimize() (basisopt.basis.molecular.MolecularBasis method), 12
optimize() (in module basisopt.opt.optimizers), 17
orbital_basis (basisopt.opt.strategies.Strategy attribute), 21
orca() (in module basisopt.api), 30
OrcaWrapper (class in basisopt.wrappers.orca), 27

P
params (basisopt.opt.strategies.Strategy attribute), 20
params (basisopt.opt.strategies.Strategy.pre attribute), 21
polarizability() (basisopt.wrappers dummy.DummyWrapper method), 27
polarizability() (basisopt.wrappers.orca.OrcaWrapper method), 28
polarizability() (basisopt.wrappers.wrapper.Wrapper method), 29
pre (basisopt.opt.strategies.Strategy attribute), 21
prop (basisopt.wrappers.wrapper.Wrapper attribute), 29
PropertyNotAvailable, 36
PropertyTest (class in basisopt.testing.test), 25
psi4() (in module basisopt.api), 30

Q
quadrupole() (basisopt.wrappers.dummy.DummyWrapper method), 27
quadrupole() (basisopt.wrappers.orca.OrcaWrapper method), 28
quadrupole() (basisopt.wrappers.wrapper.Wrapper method), 29
qz() (in module basisopt.basis.zetatools), 15
qzp() (in module basisopt.basis.zetatools), 15
qzpp() (in module basisopt.basis.zetatools), 15

R
rank_primitives() (in module basisopt.testing.rank), 24
read_json() (in module basisopt.util), 41
reduce_primitives() (in module basisopt.testing.rank), 24
reduced_mass() (basisopt.testing.dunham.DunhamTest method), 24
ReduceStrategy (class in basisopt.opt.reduce), 18
reduction_step (basisopt.opt.reduce.ReduceStrategy attribute), 19
reference (basisopt.testing.test.Test attribute), 25
register_backend() (in module basisopt.api), 30
register_quality() (in module basisopt.basis.zetatools), 15
register_test() (basisopt.basis.basis.Basis method), 10
Result (class in basisopt.containers), 33
results (basisopt.basis.basis.Basis attribute), 10
run() (basisopt.wrappers.wrapper.Wrapper method), 29
run_all() (in module basisopt.api), 30
run_all_tests() (basisopt.basis.basis.Basis method), 11
run_all_tests() (basisopt.basis.molecular.MolecularBasis method), 13
run_calculation() (in module basisopt.api), 31
run_test() (basisopt.basis.basis.Basis method), 11
run_test() (basisopt.basis.molecular.MolecularBasis method), 13

S
save() (basisopt.basis.atomic.AtomicBasis method), 9
save() (basisopt.basis.basis.Basis method), 11
save() (basisopt.basis.molecular.MolecularBasis method), 13
save() (basisopt.containers.Result method), 34
saved_basis (basisopt.opt.reduce.ReduceStrategy attribute), 18
search() (basisopt.containers.Result method), 34
set_active() (basisopt.opt.eventemper.EvenTemperedStrategy method), 16
set_active() (basisopt.opt.strategies.Strategy method), 22
set_backend() (in module basisopt.api), 31
set_basis_shells() (basisopt.opt.eventemper.EvenTemperedStrategy method), 16
set_basis_shells() (basisopt.opt.reduce.ReduceStrategy method), 20
set_dummy_atoms() (basisopt.molecule.Molecule method), 39
set_ecps() (basisopt.molecule.Molecule method), 39
set_even_tempered() (basisopt.basis.atomic.AtomicBasis method), 9
set_logger() (in module basisopt.api), 31
set_molecule_from_xyz() (basisopt.testing.test.Test method), 26
set_parallel() (in module basisopt.api), 31
set_tmp_dir() (in module basisopt.api), 31
setup() (basisopt.basis.atomic.AtomicBasis method), 9
setup() (basisopt.basis.molecular.MolecularBasis method), 13
Shell (class in basisopt.containers), 34
shell_mins (basisopt.opt.reduce.ReduceStrategy attribute), 19
shells (basisopt.opt.reduce.ReduceStrategy attribute), 19
statistics() (basisopt.containers.Result method), 34
Strategy (class in basisopt.opt.strategies), 20
string_to_config() (in module basisopt.basis.zetatools), 15
summary() (basisopt.containers.Result method), 34

target (basisopt.opt.reduce.ReduceStrategy attribute), 19
Test (class in basisopt.testing.test), 25
to_xyz() (basisopt.molecule.Molecule method), 39
trans_dipole() (basisopt.wrappers.wrapper.Wrapper method), 29
trans_quadrupole() (basisopt.wrappers.wrapper.Wrapper method), 30
tz() (in module basisopt.basis.zetatools), 15
tzp() (in module basisopt.basis.zetatools), 15
tzpp() (in module basisopt.basis.zetatools), 15

unavailable() (in module basisopt.wrappers.wrapper), 30
uncontract() (in module basisopt.basis.basis), 11
uncontract_shell() (in module basisopt.basis.basis), 11
unique_atoms() (basisopt.basis.molecular.MolecularBasis method), 13
unique_atoms() (basisopt.molecule.Molecule method), 39
unit() (in module basisopt.opt.preconditioners), 18

verify_method_string() (basisopt.wrappers.wrapper.Wrapper method), 30

which_backend() (in module basisopt.api), 31
Wrapper (class in basisopt.wrappers.wrapper), 28
write_json() (in module basisopt.util), 41