chemtools Documentation

Release 0.9.2

Lukasz Mentel

Nov 25, 2020

Contents

Chemtools is a set of modules that is intended to help with more advanced computations using common electronic structure programs.

The main to goal was to enable convenient basis set manipulations, including designing and optimizing exponents of basis sets. To achieve that there are several modules abstracting various functionalities:

Basis set object `basisset` module that contains the *`chemtools.basisset.BasisSet`*. See the *Tutorials* page for a overview of the capabilities.

Basis set optimization `basisopt` module that contains functionalities for building up basis set optimization tasks. See the *Tutorials* page for a few illustrative examples.

Calculators `chemtools.calculators.calculator` contains wrappers for several packages handling the actual electronic structure calculations. Currently there is support for:

- Dalton
- Gamess-US
- MolPro
- PSI4

Molecule *`chemtools.molecule`* is general purpose module intorducing molecule class for handling molecules and defining atomic compositions and molecular geometries

Complete basis set extrapolation *`chemtools.cbs`* contains functions for performing complete basis set (CBS) extrapolation using different approaches.

CHAPTER 1

Contents

1.1 Installation

1.1.1 Using pip

If you don't have `pip` you should probably me your life easier and get it, the installation instructions can be found here. Once you have it type

```
$ pip install chemtools
```

1.2 Tutorials

1.2.1 `BasisSet` class tutorial

`BasisSet` class provides a handy abstraction for storing and manipulating orbitals basis sets used in quantum chemistry.

Quick overview

The `BasisSet` requires some basic information

- `name`: name of the basis set as string,
- `element`: symbol or name of the element,
- `functions`: actual functions in terms of exponents and contraction coeffcients.

Internally the `functions` attribute is a dictionary with shell labels `s`, `p`, `d`, etc. as keys. A nested dictionary for each shell stores a numpy array array with exponents under key `e` and a list of contracted functions where each item is a numpy array with a custom `dtype` storing exponent indices (`idx`) and contraction coefficients (`cc`) of the functions. As an example consider the following structure:

```
{'s' : {'e'  : numpy.array([exp0, exp1, exp2, ...]),
        'cf' : [
                numpy.array([(i00, c00), (i01, c01), (i02, c02) ...], dtype=[('idx', '
↪<i4'), ('cc', '<f8')]),
                numpy.array([(i10, c10), (i11, c11), (i12, c12) ...], dtype=[('idx', '
↪<i4'), ('cc', '<f8')]),
                ...
                ]
        }
 'p' : {'e'  : numpy.array([exp0, exp1, exp2, ...]),
        'cf' : [
                numpy.array([(i00, c00), (i01, c01), (i02, c02) ...], dtype=[('idx', '
↪<i4'), ('cc', '<f8')]),
                numpy.array([(i10, c10), (i11, c11), (i12, c12) ...], dtype=[('idx', '
↪<i4'), ('cc', '<f8')]),
                ...
                ]
        }
  ...
}
```

Parsing basis sets

At present `BasisSet` can be created from the following code formats (see below for examples):

- Molpro
- Gamess(US)
- Gaussian

The basis sets can e parsed from string or from files. Here only the `from_string` will be shown but the `from_file` method has the same call sigantre with first argument being the file name.

However `BasisSet` can also be serialized into JSON and pickle formats.

- JSON
- pickle

cc-pvdz basis for Be in molpro format

[1]:
```
mpstr = '''basis={
!
! BERYLLIUM       (9s,4p,1d) -> [3s,2p,1d]
! BERYLLIUM       (9s,4p,1d) -> [3s,2p,1d]
s, BE , 2940.0000000, 441.2000000, 100.5000000, 28.4300000, 9.1690000, 3.1960000, 1.
↪1590000, 0.1811000, 0.0589000
c, 1.8, 0.0006800, 0.0052360, 0.0266060, 0.0999930, 0.2697020, 0.4514690, 0.2950740,␣
↪0.0125870
c, 1.8, -0.0001230, -0.0009660, -0.0048310, -0.0193140, -0.0532800, -0.1207230, -0.
↪1334350, 0.5307670
c, 9.9, 1
p, BE , 3.6190000, 0.7110000, 0.1951000, 0.0601800
c, 1.3, 0.0291110, 0.1693650, 0.5134580
c, 4.4, 1
```

(continues on next page)

(continued from previous page)

```
d, BE , 0.2380000
c, 1.1, 1
}'''
```

[2]:
```
from chemtools.basisset import BasisSet, merge
```

Construct a `BasisSet` instance from string

[3]:
```
pvdz = BasisSet.from_str(mpstr, fmt='molpro', name='cc-pvdz')
```

[4]:
```
print(pvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1     2940.0000000000     0.00068000    -0.00012300
    2      441.2000000000     0.00523600    -0.00096600
    3      100.5000000000     0.02660600    -0.00483100
    4       28.4300000000     0.09999300    -0.01931400
    5        9.1690000000     0.26970200    -0.05328000
    6        3.1960000000     0.45146900    -0.12072300
    7        1.1590000000     0.29507400    -0.13343500
    8        0.1811000000     0.01258700     0.53076700
Uncontracted:
    9        0.0589000000     1.00000000

================p shell=================
Contracted:
    1        3.6190000000     0.02911100
    2        0.7110000000     0.16936500
    3        0.1951000000     0.51345800
Uncontracted:
    4        0.0601800000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
```

cc-pvdz basis for Be in Gaussian format

[5]:
```
gaustr = '''****
Be     0
S   8   1.00
   2940.0000000              0.0006800
    441.2000000              0.0052360
    100.5000000              0.0266060
     28.4300000              0.0999930
      9.1690000              0.2697020
```

(continues on next page)

(continued from previous page)

```
      3.1960000              0.4514690
      1.1590000              0.2950740
      0.1811000              0.0125870
S   8   1.00
   2940.0000000             -0.0001230
    441.2000000             -0.0009660
    100.5000000             -0.0048310
     28.4300000             -0.0193140
      9.1690000             -0.0532800
      3.1960000             -0.1207230
      1.1590000             -0.1334350
      0.1811000              0.5307670
S   1   1.00
      0.0589000              1.0000000
P   3   1.00
      3.6190000              0.0291110
      0.7110000              0.1693650
      0.1951000              0.5134580
P   1   1.00
      0.0601800              1.0000000
D   1   1.00
      0.2380000              1.0000000
****'''
```

[6]:
```
gaupvdz = BasisSet.from_str(gaustr, fmt='gaussian', name='cc-pvdz')
```

[7]:
```
print(gaupvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1     2940.0000000000     0.00068000    -0.00012300
    2      441.2000000000     0.00523600    -0.00096600
    3      100.5000000000     0.02660600    -0.00483100
    4       28.4300000000     0.09999300    -0.01931400
    5        9.1690000000     0.26970200    -0.05328000
    6        3.1960000000     0.45146900    -0.12072300
    7        1.1590000000     0.29507400    -0.13343500
    8        0.1811000000     0.01258700     0.53076700
Uncontracted:
    9        0.0589000000     1.00000000

================p shell=================
Contracted:
    1        3.6190000000     0.02911100
    2        0.7110000000     0.16936500
    3        0.1951000000     0.51345800
Uncontracted:
    4        0.0601800000     1.00000000

================d shell=================
```

(continues on next page)

(continued from previous page)

```
Uncontracted:
    1        0.2380000000     1.00000000
```

cc-pvdz for Be in Gamess(US) format

[8]:
```
gamstr = '''$DATA
BERYLLIUM
S   8
  1   2940.0000000              0.0006800
  2    441.2000000              0.0052360
  3    100.5000000              0.0266060
  4     28.4300000              0.0999930
  5      9.1690000              0.2697020
  6      3.1960000              0.4514690
  7      1.1590000              0.2950740
  8      0.1811000              0.0125870
S   8
  1   2940.0000000             -0.0001230
  2    441.2000000             -0.0009660
  3    100.5000000             -0.0048310
  4     28.4300000             -0.0193140
  5      9.1690000             -0.0532800
  6      3.1960000             -0.1207230
  7      1.1590000             -0.1334350
  8      0.1811000              0.5307670
S   1
  1      0.0589000              1.0000000
P   3
  1      3.6190000              0.0291110
  2      0.7110000              0.1693650
  3      0.1951000              0.5134580
P   1
  1      0.0601800              1.0000000
D   1
  1      0.2380000              1.0000000
$END
'''
```

[9]:
```
gampvdz = BasisSet.from_str(gamstr, fmt='gamessus', name='cc-pvdz')
```

[10]:
```
print(gampvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1     2940.0000000000     0.00068000    -0.00012300
    2      441.2000000000     0.00523600    -0.00096600
    3      100.5000000000     0.02660600    -0.00483100
```

(continues on next page)

(continued from previous page)

```
    4       28.4300000000     0.09999300    -0.01931400
    5        9.1690000000     0.26970200    -0.05328000
    6        3.1960000000     0.45146900    -0.12072300
    7        1.1590000000     0.29507400    -0.13343500
    8        0.1811000000     0.01258700     0.53076700
Uncontracted:
    9        0.0589000000     1.00000000

================p shell=================
Contracted:
    1        3.6190000000     0.02911100
    2        0.7110000000     0.16936500
    3        0.1951000000     0.51345800
Uncontracted:
    4        0.0601800000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
```

Basis sets for multiple elements

If basis sets for multiple elements are given in the format compatible with EMSL BSE the parser returns a dictionary with element symbols as keys and `BasisSet` objects as values. For example below is the cc-pCVDZ basis set string in molpro format for Be, Mg, Ca as downloaded from EMSL. Remaining Gamess(US) and Gaussian formats can be parsed analogously.

[11]:
```
multi = '''
basis={
!
! BERYLLIUM       (10s,5p,1d) -> [4s,3p,1d]
! BERYLLIUM       (9s,4p,1d) -> [3s,2p,1d]
! BERYLLIUM       (1s,1p)
s, BE , 2940.0000000, 441.2000000, 100.5000000, 28.4300000, 9.1690000, 3.1960000, 1.
↪1590000, 0.1811000, 0.0589000, 1.8600000
c, 1.8, 0.0006800, 0.0052360, 0.0266060, 0.0999930, 0.2697020, 0.4514690, 0.2950740,␣
↪0.0125870
c, 1.8, -0.0001230, -0.0009660, -0.0048310, -0.0193140, -0.0532800, -0.1207230, -0.
↪1334350, 0.5307670
c, 9.9, 1
c, 10.10, 1
p, BE , 3.6190000, 0.7110000, 0.1951000, 0.0601800, 6.1630000
c, 1.3, 0.0291110, 0.1693650, 0.5134580
c, 4.4, 1
c, 5.5, 1
d, BE , 0.2380000
c, 1.1, 1
! MAGNESIUM       (13s,9p,2d) -> [5s,4p,2d]
! MAGNESIUM       (12s,8p,1d) -> [4s,3p,1d]
! MAGNESIUM       (1s,1p,1d)
s, MG , 47390.0000000, 7108.0000000, 1618.0000000, 458.4000000, 149.3000000, 53.
↪5900000, 20.7000000, 8.3840000, 2.5420000, 0.8787000, 0.1077000, 0.0399900, 3.
↪4220000
c, 1.11, 0.346023D-03, 0.268077D-02, 0.138367D-01, 0.551767D-01, 0.169660D+00, 0.
↪364703D+00, 0.406856D+00, 0.135089D+00, 0.490884D-02, 0.286460D-03, 0.2[illegible]
```

(continues on next page)

(continued from previous page)

```
c, 1.11, -0.877839D-04, -0.674725D-03, -0.355603D-02, -0.142154D-01, -0.476748D-01, -
↪0.114892D+00, -0.200676D+00, -0.341224D-01, 0.570454D+00, 0.542309D+00, 0.218128D-01
c, 1.11, 0.169628D-04, 0.129865D-03, 0.688831D-03, 0.273533D-02, 0.931224D-02, 0.
↪223265D-01, 0.411195D-01, 0.545642D-02, -0.134012D+00, -0.256176D+00, 0.605856D+00
c, 12.12, 1
c, 13.13, 1
p, MG , 179.9000000, 42.1400000, 13.1300000, 4.6280000, 1.6700000, 0.5857000, 0.
↪1311000, 0.0411200, 8.2790000
c, 1.7, 0.538161D-02, 0.392418D-01, 0.157445D+00, 0.358535D+00, 0.457226D+00, 0.
↪215918D+00, 0.664948D-02
c, 1.7, -0.865948D-03, -0.615978D-02, -0.261519D-01, -0.570647D-01, -0.873906D-01, -0.
↪122990D-01, 0.502085D+00
c, 8.8, 1
c, 9.9, 1
d, MG , 0.1870000, 3.7040000
c, 1.1, 1
c, 2.2, 1
! CALCIUM       (15s,12p,6d) -> [6s,5p,3d]
! CALCIUM       (14s,11p,5d) -> [5s,4p,2d]
! CALCIUM       (1s,1p,1d)
s, CA , 190000.7000000, 28481.4600000, 6482.7010000, 1835.8910000, 598.7243000, 215.
↪8841000, 84.0124200, 34.2248800, 10.0249700, 4.0559200, 1.0202610, 0.4268650, 0.
↪0633470, 0.0263010, 1.1143000
c, 1.13, 0.00022145, 0.00171830, 0.00892348, 0.03630183, 0.11762223, 0.28604352, 0.
↪42260708, 0.25774366, 0.02391893, -0.00495218, 0.00171779, -0.00089209, 0.00024510
c, 1.13, -0.00006453, -0.00049662, -0.00262826, -0.01066845, -0.03713509, -0.09804284,
↪ -0.20342692, -0.15244655, 0.48279406, 0.62923839, 0.06164842, -0.01479971, 0.
↪00361089
c, 1.13, 0.00002223, 0.00017170, 0.00090452, 0.00370343, 0.01283750, 0.03475459, 0.
↪07303491, 0.06100083, -0.24292928, -0.48708500, 0.56502804, 0.65574386, 0.02672894
c, 1.13, 0.00000531, 0.00004111, 0.00021568, 0.00088827, 0.00305813, 0.00837608, 0.
↪01741056, 0.01515453, -0.06207919, -0.12611803, 0.17360694, 0.37822943, -0.65964698
c, 14.14, 1
c, 15.15, 1
p, CA , 1072.0430000, 253.8439000, 81.3162600, 30.2418300, 12.1011000, 5.0225540, 1.
↪9092200, 0.7713040, 0.3005700, 0.0766490, 0.0277720, 1.5101000
c, 1.10, 0.00198166, 0.01612944, 0.07657851, 0.23269594, 0.42445210, 0.37326402, 0.
↪07868530, -0.00599927, 0.00264257, -0.00085694
c, 1.10, -0.00064891, -0.00527907, -0.02581131, -0.08062892, -0.15846552, -0.12816816,
↪ 0.25610103, 0.58724068, 0.30372561, 0.01416451
c, 1.10, 0.00013595, 0.00109420, 0.00542680, 0.01674718, 0.03389863, 0.02531183, -0.
↪05895713, -0.15876120, -0.08554523, 0.54464665
c, 11.11, 1
c, 12.12, 1
d, CA , 10.3182000, 2.5924200, 0.7617000, 0.2083800, 0.0537000, 1.3743000
c, 1.4, 0.03284900, 0.14819200, 0.31092100, 0.45219500
c, 5.5, 1
c, 6.6, 1
}'''
```

It can be parsed analogously to a basis for single element using `from_string` method or `from_file` when the basis is stored in a text file

[12]:
```
bsdict = BasisSet.from_str(multi, name='cc-pCVDZ', fmt='molpro')
```

The parsed basis set can be accessed by element symbols

[13]:
```
bsdict.keys()
```

[13]:
```
dict_keys(['Be', 'Mg', 'Ca'])
```

To test the parsed basis we can print some of the properties of the basis sets

[14]:
```
for symbol, bs in bsdict.items():
    print(symbol, bs.name, bs.element, bs.contraction_scheme())
```

```
Be cc-pCVDZ Be (10s5p1d) -> [4s3p1d] : {8 8 1 1/3 1 1/1}
Mg cc-pCVDZ Mg (13s9p2d) -> [5s4p2d] : {11 11 11 1 1/7 7 1 1/1 1}
Ca cc-pCVDZ Ca (15s12p6d) -> [6s5p3d] : {13 13 13 13 1 1/10 10 10 1 1/4 1 1}
```

Initialization of `BasisSet` from various sequences

The BasisSet exponents can be generated from following sequences:

- even tempered
- well tempered
- legendre expansion

To generate the exponents `from_sequence` classmethod is used. The arguments that have to be specified are

- `formula` describing the sequence, one of *eventemp*, *welltemp*, *legendre*, can be a single value or a list of values with the same length as `funs`
- `name` of the basis set
- `element`
- `funs` a list of 4-tuples, where the tuple elements are *shell*, *sequence name*, *number of functions*, *parameters*

even tempered

10 s functions with $\alpha = 0.5$ and $\beta = 2.0$

[15]:
```
et = BasisSet.from_sequence(name='my basis', element='Be', funs=[('s', 'et', 10, (0.5,
↪ 2.0))])
print(et)
```

```
Name                 = my basis
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Uncontracted:
    1      256.0000000000     1.00000000
    2      128.0000000000     1.00000000
    3       64.0000000000     1.00000000
    4       32.0000000000     1.00000000
    5       16.0000000000     1.00000000
    6        8.0000000000     1.00000000
    7        4.0000000000     1.00000000
    8        2.0000000000     1.00000000
```

(continues on next page)

(continued from previous page)

```
    9        1.0000000000     1.00000000
   10        0.5000000000     1.00000000
```

8 s functions with $\alpha = 0.8$, $\beta = 2.5$ and 6 p functions with $\alpha = 0.2$, $\beta = 3.0$

[16]:
```
et2 = BasisSet.from_sequence(name='my basis', element='Be',
                             funs=[('s', 'et', 8, (0.8, 2.5)), ('p', 'et', 6, (0.2, 3.
↪0))])
print(et)
```

```
Name                 = my basis
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Uncontracted:
    1      256.0000000000     1.00000000
    2      128.0000000000     1.00000000
    3       64.0000000000     1.00000000
    4       32.0000000000     1.00000000
    5       16.0000000000     1.00000000
    6        8.0000000000     1.00000000
    7        4.0000000000     1.00000000
    8        2.0000000000     1.00000000
    9        1.0000000000     1.00000000
   10        0.5000000000     1.00000000
```

well tempered

[17]:
```
wt = BasisSet.from_sequence(name='my basis', element='Be', funs=[('p', 'wt', 6, (0.5,␣
↪2.0, 0.9, 1.2))])
print(wt)
```

```
Name                 = my basis
Element              = Be
Family               = None
Kind                 = None
Functions:

================p shell=================
Uncontracted:
    1       30.4000000000     1.00000000
    2       13.7851550240     1.00000000
    3        6.2130589876     1.00000000
    4        2.7834955070     1.00000000
    5        1.2408224685     1.00000000
    6        0.5524120339     1.00000000
```

legendre

[18]:
```
leg = BasisSet.from_sequence(name='my basis', element='Be', funs=[('d', 'le', 8, (0.5,
↪ 2.0))])
print(leg)
```

```
Name                    = my basis
Element                 = Be
Family                  = None
Kind                    = None
Functions:

================d shell=================
Uncontracted:
    1       12.1824939607     1.00000000
    2        6.8796751109     1.00000000
    3        3.8850772086     1.00000000
    4        2.1939735051     1.00000000
    5        1.2389765975     1.00000000
    6        0.6996725374     1.00000000
    7        0.3951177613     1.00000000
    8        0.2231301601     1.00000000
```

direct specification of exponents

[19]:
```
de = BasisSet.from_sequence(name='my exponents', element='Be', funs=[('s', 'exp', 5,␣
↪(0.01, 0.1, 0.5, 2.0, 10.0))])
print(de)
```

```
Name                    = my exponents
Element                 = Be
Family                  = None
Kind                    = None
Functions:

================s shell=================
Uncontracted:
    1       10.0000000000     1.00000000
    2        2.0000000000     1.00000000
    3        0.5000000000     1.00000000
    4        0.1000000000     1.00000000
    5        0.0100000000     1.00000000
```

mixed

In this example a basis set is created by taking:

- 6 *s* functions generated from an even tempered sequence
- 4 *p* functions generated from a well tempered sequence
- 4 *d* functions generated from a legendre sequence

[20]:
```
basis = BasisSet.from_sequence(name='composite', element='He',
                               funs=[('s', 'et', 6, (0.5, 4.0)),
                                     ('p', 'wt', 4, (0.9, 3.0, 0.8, 1.2)),
                                     ('d', 'le', 4, (1.0, 2.5))])
print(basis)
```

```
Name                 = composite
Element              = He
Family               = None
Kind                 = None
Functions:

================s shell=================
Uncontracted:
    1      512.0000000000     1.00000000
    2      128.0000000000     1.00000000
    3       32.0000000000     1.00000000
    4        8.0000000000     1.00000000
    5        2.0000000000     1.00000000
    6        0.5000000000     1.00000000

================p shell=================
Uncontracted:
    1       43.7400000000     1.00000000
    2       12.6882653049     1.00000000
    3        3.6401946084     1.00000000
    4        1.0364144910     1.00000000

================d shell=================
Uncontracted:
    1       33.1154519587     1.00000000
    2        6.2547009519     1.00000000
    3        1.1813604129     1.00000000
    4        0.2231301601     1.00000000
```

Conversion to formats of different quantum chemistry programs

The following basis set formats are supported:

- Aces II/Cfour
- Dalton
- Gamess(US)
- Gaussian
- Molpro
- NWChem

In addition a converter to LaTeX is also available.

If you would like some other formats to be included please consider submitting an issue describing the request.

Gaussian format

[21]:
```
print(pvdz.to_gaussian())
```

```
****
Be     0
S   8   1.00
     2940.0000000000     0.00068000
      441.2000000000     0.00523600
      100.5000000000     0.02660600
       28.4300000000     0.09999300
        9.1690000000     0.26970200
        3.1960000000     0.45146900
        1.1590000000     0.29507400
        0.1811000000     0.01258700
S   8   1.00
     2940.0000000000    -0.00012300
      441.2000000000    -0.00096600
      100.5000000000    -0.00483100
       28.4300000000    -0.01931400
        9.1690000000    -0.05328000
        3.1960000000    -0.12072300
        1.1590000000    -0.13343500
        0.1811000000     0.53076700
S   1   1.00
        0.0589000000     1.00000000
P   3   1.00
        3.6190000000     0.02911100
        0.7110000000     0.16936500
        0.1951000000     0.51345800
P   1   1.00
        0.0601800000     1.00000000
D   1   1.00
        0.2380000000     1.00000000
****
```

NwChem format

[22]:
```
print(pvdz.to_nwchem())
```

```
BASIS "ao basis" PRINT
Be s
     2940.0000000000     0.00068000    -0.00012300
      441.2000000000     0.00523600    -0.00096600
      100.5000000000     0.02660600    -0.00483100
       28.4300000000     0.09999300    -0.01931400
        9.1690000000     0.26970200    -0.05328000
        3.1960000000     0.45146900    -0.12072300
        1.1590000000     0.29507400    -0.13343500
        0.1811000000     0.01258700     0.53076700
Be s
        0.0589000000     1.00000000
Be p
        3.6190000000     0.02911100
        0.7110000000     0.16936500
```

(continues on next page)

(continued from previous page)

```
        0.1951000000     0.51345800
Be p
        0.0601800000     1.00000000
Be d
        0.2380000000     1.00000000
END
```

Cfour/Acesll format

[23]:
```
print(pvdz.to_cfour(comment="my comment"))
```

```
Be:cc-pvdz
my comment

  3
    0    1    2
    3    2    1
    9    4    1

  2940.00000000   441.20000000   100.50000000    28.43000000     9.16900000
     3.19600000     1.15900000     0.18110000     0.05890000

     0.00068000    -0.00012300     0.00000000
     0.00523600    -0.00096600     0.00000000
     0.02660600    -0.00483100     0.00000000
     0.09999300    -0.01931400     0.00000000
     0.26970200    -0.05328000     0.00000000
     0.45146900    -0.12072300     0.00000000
     0.29507400    -0.13343500     0.00000000
     0.01258700     0.53076700     0.00000000
     0.00000000     0.00000000     1.00000000

     3.61900000     0.71100000     0.19510000     0.06018000

     0.02911100     0.00000000
     0.16936500     0.00000000
     0.51345800     0.00000000
     0.00000000     1.00000000

     0.23800000

     1.00000000
```

Dalton format

[24]:
```
print(pvdz.to_dalton())
```

```
! cc-pvdz
! s functions
```

(continues on next page)

(continued from previous page)

```
H   9   3
     2940.0000000000        0.0006800000       -0.0001230000        0.0000000000
      441.2000000000        0.0052360000       -0.0009660000        0.0000000000
      100.5000000000        0.0266060000       -0.0048310000        0.0000000000
       28.4300000000        0.0999930000       -0.0193140000        0.0000000000
        9.1690000000        0.2697020000       -0.0532800000        0.0000000000
        3.1960000000        0.4514690000       -0.1207230000        0.0000000000
        1.1590000000        0.2950740000       -0.1334350000        0.0000000000
        0.1811000000        0.0125870000        0.5307670000        0.0000000000
        0.0589000000        0.0000000000        0.0000000000        1.0000000000
! p functions
H   4   2
        3.6190000000        0.0291110000        0.0000000000
        0.7110000000        0.1693650000        0.0000000000
        0.1951000000        0.5134580000        0.0000000000
        0.0601800000        0.0000000000        1.0000000000
! d functions
H   1   1
        0.2380000000        1.0000000000
```

Gamess(US) format

[25]:
```
print(pvdz.to_gamessus())
```

```
S  8
  1     2940.0000000000     0.00068000
  2      441.2000000000     0.00523600
  3      100.5000000000     0.02660600
  4       28.4300000000     0.09999300
  5        9.1690000000     0.26970200
  6        3.1960000000     0.45146900
  7        1.1590000000     0.29507400
  8        0.1811000000     0.01258700
S  8
  1     2940.0000000000    -0.00012300
  2      441.2000000000    -0.00096600
  3      100.5000000000    -0.00483100
  4       28.4300000000    -0.01931400
  5        9.1690000000    -0.05328000
  6        3.1960000000    -0.12072300
  7        1.1590000000    -0.13343500
  8        0.1811000000     0.53076700
S  1
  1        0.0589000000     1.00000000
P  3
  1        3.6190000000     0.02911100
  2        0.7110000000     0.16936500
  3        0.1951000000     0.51345800
P  1
  1        0.0601800000     1.00000000
D  1
  1        0.2380000000     1.00000000
```

Molpro format

[26]:
```
print(pvdz.to_molpro(withpars=True))
```

```
basis={
s, Be, 2940.0000000000, 441.2000000000, 100.5000000000, 28.4300000000, 9.1690000000,␣
↪3.1960000000, 1.1590000000, 0.1811000000, 0.0589000000
c, 1.8, 0.00068000, 0.00523600, 0.02660600, 0.09999300, 0.26970200, 0.45146900, 0.
↪29507400, 0.01258700
c, 1.8, -0.00012300, -0.00096600, -0.00483100, -0.01931400, -0.05328000, -0.12072300,␣
↪-0.13343500, 0.53076700
c, 9.9, 1.00000000
p, Be, 3.6190000000, 0.7110000000, 0.1951000000, 0.0601800000
c, 1.3, 0.02911100, 0.16936500, 0.51345800
c, 4.4, 1.00000000
d, Be, 0.2380000000
c, 1.1, 1.00000000
}
```

LaTeX format

[27]:
```
print(pvdz.to_latex())
```

```
\begin{tabular}{rrrr}
No. & \multicolumn{1}{c}{Exponent} & \multicolumn{2}{c}{Coefficients } \\
\hline
\multicolumn{4}{c}{ s shell } \\ \hline
    1 &      2940.0000000000 &       0.00068000 &      -0.00012300 \\
    2 &       441.2000000000 &       0.00523600 &      -0.00096600 \\
    3 &       100.5000000000 &       0.02660600 &      -0.00483100 \\
    4 &        28.4300000000 &       0.09999300 &      -0.01931400 \\
    5 &         9.1690000000 &       0.26970200 &      -0.05328000 \\
    6 &         3.1960000000 &       0.45146900 &      -0.12072300 \\
    7 &         1.1590000000 &       0.29507400 &      -0.13343500 \\
    8 &         0.1811000000 &       0.01258700 &       0.53076700 \\
    9 &         0.0589000000 & \\
\hline
\multicolumn{4}{c}{ p shell } \\ \hline
    1 &         3.6190000000 &       0.02911100 \\
    2 &         0.7110000000 &       0.16936500 \\
    3 &         0.1951000000 &       0.51345800 \\
    4 &         0.0601800000 & \\
\hline
\multicolumn{4}{c}{ d shell } \\ \hline
    1 &         0.2380000000 & \\
\end{tabular}
```

Useful tools for working with `BasisSet`

Attributes

[28]:
```
pvdz.name
```

```
[28]: 'cc-pvdz'
```

```
[29]: pvdz.element
[29]: 'Be'
```

Inspection methods describing the `BasisSet`

```
[30]: pvdz.contraction_scheme()
[30]: '(9s4p1d) -> [3s2p1d] : {8 8 1/3 1/1}'
```

```
[31]: pvdz.contraction_type()
[31]: 'unknown'
```

Number of contracted functions per shell

```
[32]: pvdz.contractions_per_shell()
[32]: [3, 2, 1]
```

Contraction matrix for a given shell, where rows numbers correspond to the exponent indices and column contain the conntraction coefficients for each function.

```
[33]: pvdz.contraction_matrix('s')
[33]: array([[  6.80000000e-04,  -1.23000000e-04,   0.00000000e+00],
             [  5.23600000e-03,  -9.66000000e-04,   0.00000000e+00],
             [  2.66060000e-02,  -4.83100000e-03,   0.00000000e+00],
             [  9.99930000e-02,  -1.93140000e-02,   0.00000000e+00],
             [  2.69702000e-01,  -5.32800000e-02,   0.00000000e+00],
             [  4.51469000e-01,  -1.20723000e-01,   0.00000000e+00],
             [  2.95074000e-01,  -1.33435000e-01,   0.00000000e+00],
             [  1.25870000e-02,   5.30767000e-01,   0.00000000e+00],
             [  0.00000000e+00,   0.00000000e+00,   1.00000000e+00]])
```

```
[34]: pvdz.contraction_matrix('p')
[34]: array([[ 0.029111,  0.      ],
             [ 0.169365,  0.      ],
             [ 0.513458,  0.      ],
             [ 0.      ,  1.      ]])
```

Calculate the total number of functions in the basis set. By default the number is calculated assuming spherical harmonics, namely 5d functions, 7f functions, etc.

```
[35]: pvdz.nf()
[35]: 14
```

The number of functions assuming cartesian gaussians, namely 6d components, 10f components, etc., can be calcualted by passing `spherical=False` argument.

```
[36]: pvdz.nf(spherical=False)
```

[36]:
```
15
```

Calculate the number of primitive functions assuming sphrical or cartesian gaussians.

[37]:
```
pvdz.nprimitive()
```

[37]:
```
26
```

[38]:
```
pvdz.nprimitive(spherical=False)
```

[38]:
```
27
```

[39]:
```
pvdz.primitives_per_shell()
```

[39]:
```
[9, 4, 1]
```

[40]:
```
pvdz.primitives_per_contraction()
```

[40]:
```
[[8, 8, 1], [3, 1], [1]]
```

If you want to extract just the exponents you can use the `get_exponents` method, that by default returns a `dict` of exponents in each shell.

[41]:
```
pvdz.get_exponents()
```

[41]:
```
{'d': array([ 0.238]),
 'p': array([ 3.619  ,  0.711  ,  0.1951 ,  0.06018]),
 's': array([  2.94000000e+03,   4.41200000e+02,   1.00500000e+02,
          2.84300000e+01,   9.16900000e+00,   3.19600000e+00,
          1.15900000e+00,   1.81100000e-01,   5.89000000e-02])}
```

If you want all exponents in one array pass the `asdict=False` argument

[42]:
```
pvdz.get_exponents(asdict=False)
```

[42]:
```
array([  2.94000000e+03,   4.41200000e+02,   1.00500000e+02,
         2.84300000e+01,   9.16900000e+00,   3.61900000e+00,
         3.19600000e+00,   1.15900000e+00,   7.11000000e-01,
         2.38000000e-01,   1.95100000e-01,   1.81100000e-01,
         6.01800000e-02,   5.89000000e-02])
```

Sorting

Each shell of the `BasisSet` can be sorted with respect to the exponents, the default order is descending

[43]:
```
pvdz.sort()
print(pvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1     2940.0000000000     0.00068000    -0.00012300
```

(continues on next page)

(continued from previous page)

```
    2      441.2000000000     0.00523600    -0.00096600
    3      100.5000000000     0.02660600    -0.00483100
    4       28.4300000000     0.09999300    -0.01931400
    5        9.1690000000     0.26970200    -0.05328000
    6        3.1960000000     0.45146900    -0.12072300
    7        1.1590000000     0.29507400    -0.13343500
    8        0.1811000000     0.01258700     0.53076700
Uncontracted:
    9        0.0589000000     1.00000000

================p shell=================
Contracted:
    1        3.6190000000     0.02911100
    2        0.7110000000     0.16936500
    3        0.1951000000     0.51345800
Uncontracted:
    4        0.0601800000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
```

the basis set was already sorted so there is no difference but we can also sort in ascending order to see the result

[44]:
```
pvdz.sort(reverse=True)
print(pvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1        0.1811000000     0.01258700     0.53076700
    2        1.1590000000     0.29507400    -0.13343500
    3        3.1960000000     0.45146900    -0.12072300
    4        9.1690000000     0.26970200    -0.05328000
    5       28.4300000000     0.09999300    -0.01931400
    6      100.5000000000     0.02660600    -0.00483100
    7      441.2000000000     0.00523600    -0.00096600
    8     2940.0000000000     0.00068000    -0.00012300
Uncontracted:
    9        0.0589000000     1.00000000

================p shell=================
Contracted:
    1        0.1951000000     0.51345800
    2        0.7110000000     0.16936500
    3        3.6190000000     0.02911100
Uncontracted:
    4        0.0601800000     1.00000000

================d shell=================
Uncontracted:
```

(continues on next page)

(continued from previous page)

```
    1        0.2380000000     1.00000000
```

Normalization of the contraction coefficients

To check if the contraction coefficients for each function are properly normalized to unity the `normalized` method can be called, which returns a list of tuples with the shell, contracted function index and the current normalization.

[45]:
```
pvdz.normalization()
```

[45]:
```
[('s', 0, 1.0013033713220225),
 ('s', 1, 0.23915929699104757),
 ('s', 2, 1.0),
 ('p', 0, 0.40825721905456636),
 ('p', 1, 1.0),
 ('d', 0, 1.0)]
```

To normalize the contracted functions simply call the `normalize` method

[46]:
```
pvdz.normalize()
```

to verify we can call the check the normalization again

[47]:
```
pvdz.normalization()
```

[47]:
```
[('s', 0, 0.99999999999999978),
 ('s', 1, 1.0),
 ('s', 2, 1.0),
 ('p', 0, 1.0),
 ('p', 1, 1.0),
 ('d', 0, 1.0)]
```

Uncontracting basis sets

The basis sets can be easily uncontract by using the `uncontract` method. By default the basis set is uncontracted in-place but the `copy=True` can be specified to obtain an uncotracted copy of the basis set without altering the original.

[48]:
```
upvdz = pvdz.uncontract(copy=True)
print(upvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Uncontracted:
    1        0.0589000000     1.00000000
    2        0.1811000000     1.00000000
    3        1.1590000000     1.00000000
    4        3.1960000000     1.00000000
    5        9.1690000000     1.00000000
```

(continues on next page)

(continued from previous page)

```
    6       28.4300000000     1.00000000
    7      100.5000000000     1.00000000
    8      441.2000000000     1.00000000
    9     2940.0000000000     1.00000000

================p shell=================
Uncontracted:
    1        0.0601800000     1.00000000
    2        0.1951000000     1.00000000
    3        0.7110000000     1.00000000
    4        3.6190000000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
```

Shell overlap

To calculate the overlap matrix between the functions in a given shell you can use the `shell_overlap` method

[49]:
```
pvdz.shell_overlap('s')
```

[49]:
```
array([[ 1.        , -0.20056477,  0.17519428],
       [-0.20056477,  1.        ,  0.74780722],
       [ 0.17519428,  0.74780722,  1.        ]])
```

Adding basis sets

As an example diffuse functions will be added to the existing `BasisSet` object with cc-pVDZ basis producing the aug-cc-pVDZ basis. The agumented functions are first parsed from a string to a `BasisSet` object

[50]:
```
augstr = '''! aug-cc-pVDZ

basis={

s,Be,1.790000E-02;
c,1.1,1.000000E+00;

p,Be,1.110000E-02;
c,1.1,1.000000E+00;

d,Be,7.220000E-02;
c,1.1,1.000000E+00;

}'''
```

[51]:
```
aug = BasisSet.from_str(augstr, fmt='molpro', name='aug functions')
```

[52]:
```
print(aug)
```

```
Name                 = aug functions
Element              = Be
```

(continues on next page)

(continued from previous page)

```
Family               = None
Kind                 = None
Functions:

================s shell=================
Uncontracted:
    1        0.0179000000     1.00000000

================p shell=================
Uncontracted:
    1        0.0111000000     1.00000000

================d shell=================
Uncontracted:
    1        0.0722000000     1.00000000
```

Addition of two `BasisSet` object can be done simply by using the + operator. The functions from the second argument are then added to the first argument. The order of the arguments is important since remaining attributes (name, element, family) are copied from the first argument. In the example below the resulting `apvdz` object retained the `name` and `element` attributes from the first argument namely `pvdz` object. If the addition was done in reverse order (`aug + pvdz`) the `name` of the result would be *aug functions*.

[53]:
```
apvdz = pvdz + aug
print(apvdz)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
Functions:

================s shell=================
Contracted:
    1        0.1811000000     0.01257881      1.08532618
    2        1.1590000000     0.29488189     -0.27285136
    3        3.1960000000     0.45117507     -0.24685753
    4        9.1690000000     0.26952641     -0.10894833
    5       28.4300000000     0.09992790     -0.03949377
    6      100.5000000000     0.02658868     -0.00987855
    7      441.2000000000     0.00523259     -0.00197530
    8     2940.0000000000     0.00067956     -0.00025151
Uncontracted:
    9        0.0589000000     1.00000000
   10        0.0179000000     1.00000000

================p shell=================
Contracted:
    1        0.1951000000     0.80359641
    2        0.7110000000     0.26506765
    3        3.6190000000     0.04556068
Uncontracted:
    4        0.0601800000     1.00000000
    5        0.0111000000     1.00000000

================d shell=================
Uncontracted:
```

(continues on next page)

(continued from previous page)

```
    1         0.2380000000      1.00000000
    2         0.0722000000      1.00000000
```

Serialization

Currenly there are two methods supported for serializing `BasisSet` objects: JSON and pickle.

The default is chosen to be JSON since it provides more flexibility and is human readable.

JSON serialization

To serialize a `BasisSet` to json there is a `to_json` method

[54]:
```
jsonstr = apvdz.to_json(indent=4)
print(jsonstr)
```

```
{
    "name": "cc-pvdz",
    "element": "Be",
    "family": null,
    "kind": null,
    "functions": {
        "s": {
            "e": {
                "data": [
                    0.0589,
                    0.1811,
                    1.159,
                    3.196,
                    9.169,
                    28.43,
                    100.5,
                    441.2,
                    2940.0,
                    0.0179
                ],
                "dtype": "float64"
            },
            "cf": [
                {
                    "data": [
                        [
                            1,
                            0.01257880524232446
                        ],
                        [
                            2,
                            0.29488189227565326
                        ],
                        [
                            3,
                            0.45117507141868446
                        ],
```

(continues on next page)

(continued from previous page)


```
                    [
                        4,
                        0.26952641069876787
                    ],
                    [
                        5,
                        0.0999278996262612
                    ],
                    [
                        6,
                        0.026588678182035797
                    ],
                    [
                        7,
                        0.005232591105808443
                    ],
                    [
                        8,
                        0.000679557286468629
                    ]
                ],
                "dtype": "CFDTYPE"
            },
            {
                "data": [
                    [
                        1,
                        1.08532617991957
                    ],
                    [
                        2,
                        -0.2728513619301272
                    ],
                    [
                        3,
                        -0.2468575333779799
                    ],
                    [
                        4,
                        -0.10894833112479618
                    ],
                    [
                        5,
                        -0.039493770032738615
                    ],
                    [
                        6,
                        -0.009878554573271215
                    ],
                    [
                        7,
                        -0.001975301949447318
                    ],
                    [
                        8,
                        -0.00025151360225882
                    ]
```


(continues on next page)

(continued from previous page)


```
                ],
                "dtype": "CFDTYPE"
            },
            {
                "data": [
                    [
                        0,
                        1.0
                    ]
                ],
                "dtype": "CFDTYPE"
            },
            {
                "data": [
                    [
                        9,
                        1.0
                    ]
                ],
                "dtype": "CFDTYPE"
            }
        ]
    },
    "p": {
        "e": {
            "data": [
                0.06018,
                0.1951,
                0.711,
                3.619,
                0.0111
            ],
            "dtype": "float64"
        },
        "cf": [
            {
                "data": [
                    [
                        1,
                        0.8035964108391421
                    ],
                    [
                        2,
                        0.2650676513400732
                    ],
                    [
                        3,
                        0.04556067899601967
                    ]
                ],
                "dtype": "CFDTYPE"
            },
            {
                "data": [
                    [
                        0,
                        1.0
```


(continues on next page)

(continued from previous page)


```
                        ]
                    ],
                    "dtype": "CFDTYPE"
                },
                {
                    "data": [
                        [
                            4,
                            1.0
                        ]
                    ],
                    "dtype": "CFDTYPE"
                }
            ]
        },
        "d": {
            "e": {
                "data": [
                    0.238,
                    0.0722
                ],
                "dtype": "float64"
            },
            "cf": [
                {
                    "data": [
                        [
                            0,
                            1.0
                        ]
                    ],
                    "dtype": "CFDTYPE"
                },
                {
                    "data": [
                        [
                            1,
                            1.0
                        ]
                    ],
                    "dtype": "CFDTYPE"
                }
            ]
        }
    },
    "info": null
}
```


`BasisSet` can also be created from a JSON string with the `from_json` method

[55]:
```
bsfromjson = BasisSet.from_json(jsonstr)
print(bsfromjson)
```

```
Name                 = cc-pvdz
Element              = Be
Family               = None
Kind                 = None
```

(continues on next page)

(continued from previous page)

```
Functions:

================s shell=================
Contracted:
    1        0.1811000000     0.01257881     1.08532618
    2        1.1590000000     0.29488189    -0.27285136
    3        3.1960000000     0.45117507    -0.24685753
    4        9.1690000000     0.26952641    -0.10894833
    5       28.4300000000     0.09992790    -0.03949377
    6      100.5000000000     0.02658868    -0.00987855
    7      441.2000000000     0.00523259    -0.00197530
    8     2940.0000000000     0.00067956    -0.00025151
Uncontracted:
    9        0.0589000000     1.00000000
   10        0.0179000000     1.00000000

================p shell=================
Contracted:
    1        0.1951000000     0.80359641
    2        0.7110000000     0.26506765
    3        3.6190000000     0.04556068
Uncontracted:
    4        0.0601800000     1.00000000
    5        0.0111000000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
    2        0.0722000000     1.00000000
```

pickle serialization

Analogously to JSON, to serialize to the `BasisSet` to pickle there is a `to_pickle` method which accepts a file anem argument and writes a pickle file to disk.

[56]:
```
apvdz.to_pickle('aug-cc-pvdz.bas')
```

The pickle can be read back into the `BasisSet` object by `read_pickle` method

[60]:
```
bsfrompickle = BasisSet.from_pickle('aug-cc-pvdz.bas')
print(bsfrompickle)
```

```
Name                = cc-pvdz
Element             = Be
Family              = None
Kind                = None
Functions:

================s shell=================
Contracted:
    1        0.1811000000     0.01257881     1.08532618
    2        1.1590000000     0.29488189    -0.27285136
    3        3.1960000000     0.45117507    -0.24685753
    4        9.1690000000     0.26952641    -0.10894833
```

(continues on next page)

(continued from previous page)

```
    5       28.4300000000     0.09992790    -0.03949377
    6      100.5000000000     0.02658868    -0.00987855
    7      441.2000000000     0.00523259    -0.00197530
    8     2940.0000000000     0.00067956    -0.00025151
Uncontracted:
    9        0.0589000000     1.00000000
   10        0.0179000000     1.00000000

================p shell=================
Contracted:
    1        0.1951000000     0.80359641
    2        0.7110000000     0.26506765
    3        3.6190000000     0.04556068
Uncontracted:
    4        0.0601800000     1.00000000
    5        0.0111000000     1.00000000

================d shell=================
Uncontracted:
    1        0.2380000000     1.00000000
    2        0.0722000000     1.00000000
```

Completeness profiles

To visually inspect the quality of the basis set, completeness profiles can be calculated [Chong 1995, Lehtola 2014]. The completeness profile calculation is also implemented in the kruununhaka basis set tool kit.

[63]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd

%matplotlib inline
```

To specify the range of scanning exponents for the basis set we'll use the grid of equidistant points in the log space

[64]:
```
zetas = np.logspace(-10, 10, num=2000, endpoint=True, base=10.0)
```

The `completeness_profile` method calculates the profile and returns a numpy array with profile values per shell stored in columns. For convenience we'll convert the numpy array to a `DataFrame`

[65]:
```
cp = pvdz.completeness_profile(zetas)
df = pd.DataFrame(cp, columns=pvdz.functions.keys())
```

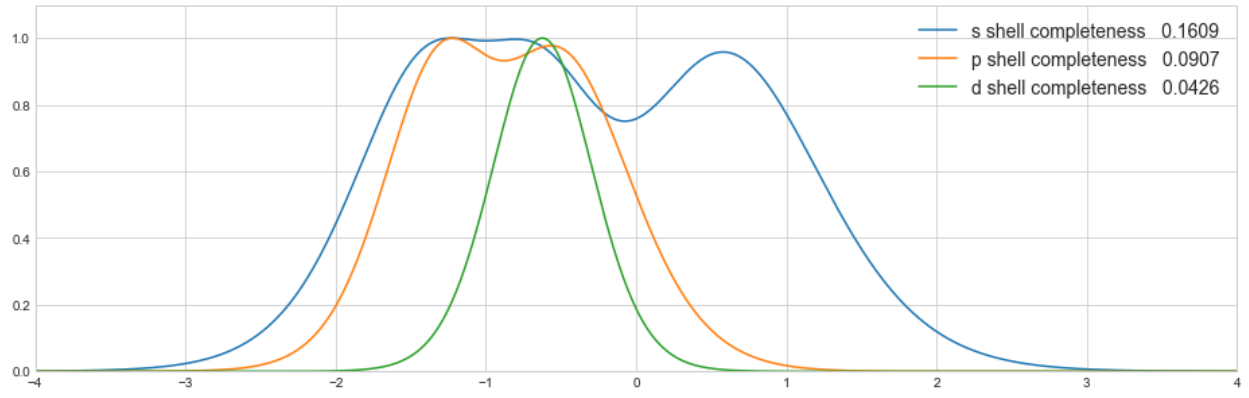
Now we can plot the profiles for each shell together

[66]:
```
sns.set_style('whitegrid')
plt.figure(figsize=(16, 5))
for shell in df.columns:
    c = df[shell].sum()/df[shell].size
    plt.plot(np.log10(zetas), df[shell], label='{} shell completeness {:8.4f}'.
↪format(shell, c))
plt.legend(loc='best', frameon=False, fontsize='14')
plt.xlim(-4, 4)
```

(continues on next page)

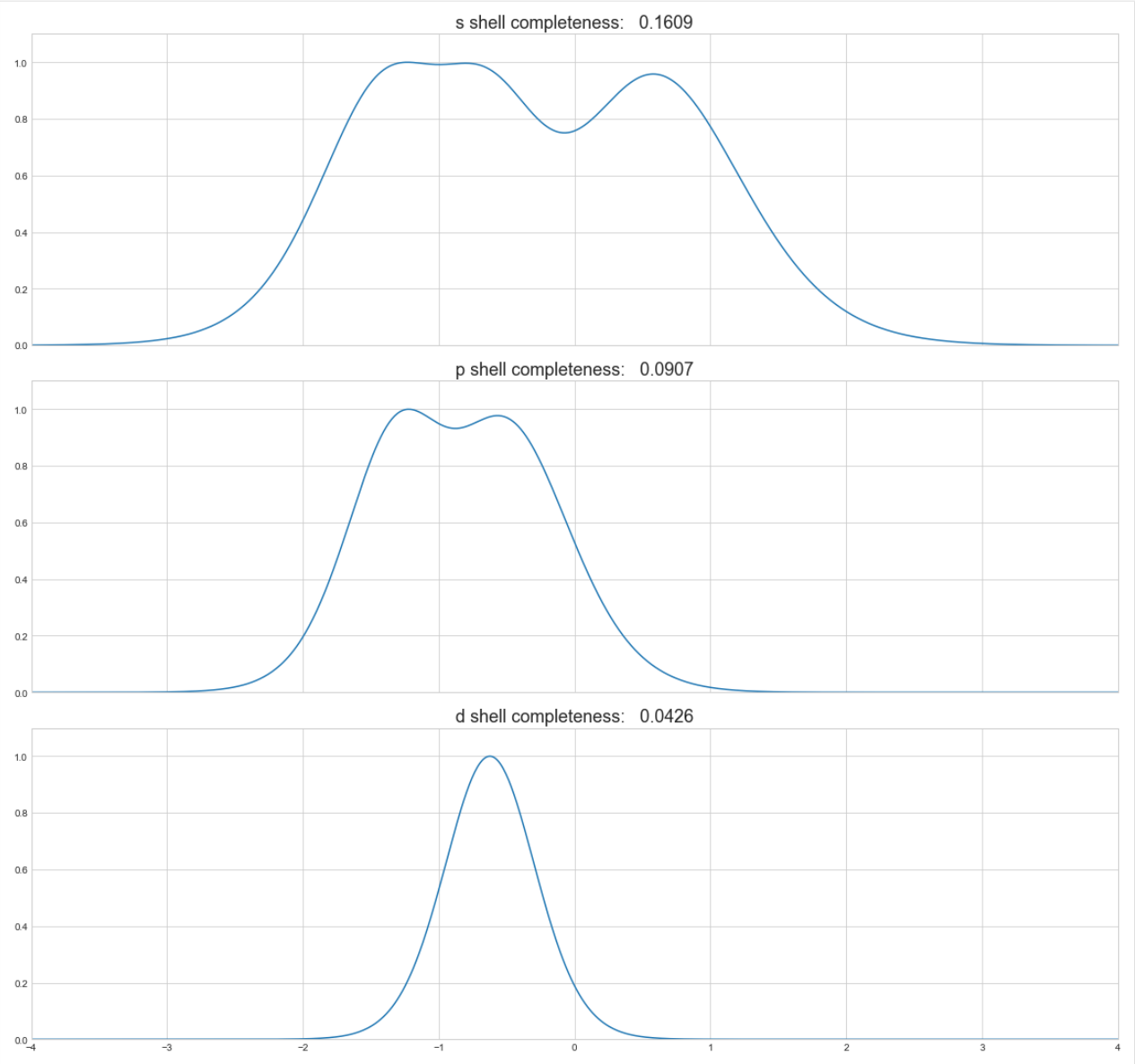
(continued from previous page)

```
plt.ylim(0.0, 1.1)
plt.show()
```



or separately

[67]:
```
f, axes = plt.subplots(df.shape[1], sharex=True, figsize=(16, 5*df.shape[1]))
for axis, shell in zip(axes, df.columns):
    axis.plot(np.log10(zetas), df[shell])
    axis.set_title('{} shell completeness: {:8.4f}'.format(shell, df[shell].sum()/
↪df[shell].size), fontsize=18)
    axis.set_ylim(0.0, 1.1)
plt.xlim(-4, 4)
plt.tight_layout()
```



[68]:
```
%version_information chemtools, mendeleev, scipy, numpy, matplotlib, seaborn
```

[68]:

Software	Version
Python	3.6.3 64bit [GCC 7.2.0]
IPython	6.2.1
OS	Linux 4.9.0 4 amd64 x86_64 with debian 9.1
chemtools	0.8.4
mendeleev	0.4.0
scipy	1.0.0
numpy	1.13.3
matplotlib	2.1.0
seaborn	0.8.1
Mon Nov 27 14:11:32 2017 CET	

1.2.2 Basis set optimization with chemtools

In this tutorial we will go over a few examples illustrating how to use the `chemtools` package to optimize the exponents of orbital basis sets using various scenarios.

First some important imports:

[1]:
```
from chemtools.basisset import BasisSet
from chemtools.basisopt import BSOptimizer
from chemtools.molecule import Molecule
from chemtools.calculators.molpro import Molpro
```

Define the program that will be used to perform the energy calculations, in this particular case it will be Molpro.

[2]:
```
mp = Molpro(exevar="MOLPRO_EXE", runopts=["-s", "-n", "1", "-d", "."])
```

Optimization of even tempered parameters at the Hartree-Fock level for Be atom

In the first example we will optimize the *s* exponents for the HF calculations.

define the system for which the optimization will be performed

[3]:
```
be = Molecule('Be', atoms=[('Be',)])
```

template for the input file to be used in sigle point calculations

[4]:
```
templ =  '''***,be
memory,100,m

%geometry

%basis

gthresh,energy=1.0e-9
{rhf; wf,4,1,0}

'''
```

[5]:
```
bso = BSOptimizer(objective='hf total energy', template=templ, code=mp, mol=be,
                  fsopt={'Be' : [('s', 'et', 8, (0.1, 2.0)),],}, verbose=False)
```

[6]:
```
bso.run()
```

```
Script name : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages/
↪ipykernel_launcher.py
Workdir     : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages
Start time  : 2017-11-27 14:56:58.315938
================================================================================
============================STARTING OPTIMIZATION===============================
================================================================================

====================================CODE========================================
<Molpro(
        name=Molpro,
        molpropath=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin,
        executable=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin/molpro,
```

(continues on next page)

(continued from previous page)

```
        scratch=/home/lmentel/scratch,
        runopts=['-s', '-n', '1', '-d', '.'],
)>
====================================MOL=======================================
Name: Be          Charge: 0          Multiplicty: 1          Electrons: 4
Atoms:
Element    Nuclear Charge              x              y              z
Be                   4.00              0.00000        0.00000        0.00000
==================================OPTALG======================================
{'jacob': None,
 'method': 'Nelder-Mead',
 'options': {'disp': True, 'maxiter': 100},
 'tol': 0.0001}Optimization terminated successfully.
         Current function value: -14.566522
         Iterations: 39
         Function evaluations: 73
 final_simplex: (array([[ 0.07025538,  3.55536302],
       [ 0.07025056,  3.55544753],
       [ 0.07025069,  3.55534628]]), array([-14.56652238, -14.56652238, -14.
↪56652238]))
           fun: -14.566522375296
       message: 'Optimization terminated successfully.'
          nfev: 73
           nit: 39
        status: 0
       success: True
             x: array([ 0.07025538,  3.55536302])Elapsed time :               18.503␣
↪sec
```

We can retrieve the optimized basis directly from the optimized through:

[8]:
```
bso.get_basis()
```

[8]:
```
{'Be': <BasisSet(
        name                 = None
        element              = None
        family               = None
        kind                 = None

 ================s shell=================
 Uncontracted:
     1      504.5070405637     1.00000000
     2      141.9002891736     1.00000000
     3       39.9116175763     1.00000000
     4       11.2257503268     1.00000000
     5        3.1574132559     1.00000000
     6        0.8880705680     1.00000000
     7        0.2497833732     1.00000000
     8        0.0702553781     1.00000000
 )>}
```

The raw optimization results are also available

[9]:
```
bso.result
```

[9]:
```
 final_simplex: (array([[ 0.07025538,  3.55536302],
       [ 0.07025056,  3.55544753],
       [ 0.07025069,  3.55534628]]), array([-14.56652238, -14.56652238, -14.
↪56652238]))
```

(continues on next page)

(continued from previous page)

```
           fun: -14.566522375296
       message: 'Optimization terminated successfully.'
          nfev: 73
           nit: 39
        status: 0
       success: True
             x: array([ 0.07025538,  3.55536302])
```

Optimization of diffuse functions for *cc-pvdz* for Be

Since Be atom doesn't form a stable negative ion diffuse functions for Be are optimized for BeH-

[10]:
```
beh = Molecule(name="BeH-", atoms=[('Be',), ('H', (0.0, 0.0, 2.724985))], sym="cnv 2",
↪ charge=-1, multiplicity=1)
```

[11]:
```
bsstr = '''basis={
s,Be,2.940000E+03,4.412000E+02,1.005000E+02,2.843000E+01,9.169000E+00,3.196000E+00,1.
↪159000E+00,1.811000E-01,5.890000E-02;
c,1.9,6.800000E-04,5.236000E-03,2.660600E-02,9.999300E-02,2.697020E-01,4.514690E-01,2.
↪950740E-01,1.258700E-02,-3.756000E-03;
c,1.9,-1.230000E-04,-9.660000E-04,-4.831000E-03,-1.931400E-02,-5.328000E-02,-1.
↪207230E-01,-1.334350E-01,5.307670E-01,5.801170E-01;
c,9.9,1.000000E+00;

p,Be,3.619000E+00,7.110000E-01,1.951000E-01,6.018000E-02;
c,1.4,2.911100E-02,1.693650E-01,5.134580E-01,4.793380E-01;
c,4.4,1.000000E+00;

d,Be,2.354000E-01;
c,1.1,1.000000E+00;

s, H , 13.0100000, 1.9620000, 0.4446000, 0.1220000, 0.0297400
c, 1.3, 0.0196850, 0.1379770, 0.4781480
c, 4.4, 1
c, 5.5, 1
p, H , 0.7270000, 0.1410000
c, 1.1, 1
c, 2.2, 1
}
'''
diffusetmp =  '''***,be
memory,100,m

%geometry

%basis

%core

gthresh,energy=1.0e-9
{rhf; wf,6,1,0}
cisd

'''
```

[12]:
```
bsd = BasisSet.from_str(bsstr, fmt='molpro', name='cc-pvdz')
```

[13]:
```
difffs = {'Be' : [('s', 'exp', 1, (0.02,)), ('p', 'exp', 1, (0.01,)), ('d', 'exp', 1,␣
↪(0.07,))]}
```

[14]:
```
bso = BSOptimizer(objective='cisd total energy', template=diffusetmp, code=mp,␣
↪mol=beh,
                  fsopt=difffs, staticbs=bsd, core=[1,0,0,0,0,0,0,0], verbose=False)
```

[15]:
```
bso.run()
```

```
Script name : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages/
↪ipykernel_launcher.py
Workdir     : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages
Start time  : 2017-11-27 15:00:26.887764
================================================================================
============================STARTING OPTIMIZATION===============================
================================================================================

====================================CODE====================================
<Molpro(
        name=Molpro,
        molpropath=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin,
        executable=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin/molpro,
        scratch=/home/lmentel/scratch,
        runopts=['-s', '-n', '1', '-d', '.'],
)>
====================================MOL=====================================
Name: BeH-       Charge: -1         Multiplicity: 1          Electrons: 6
Atoms:
Element    Nuclear Charge              x              y              z
Be                   4.00              0.00000        0.00000        0.00000
H                    1.00              0.00000        0.00000        2.72499
==================================OPTALG====================================
{'jacob': None,
 'method': 'Nelder-Mead',
 'options': {'disp': True, 'maxiter': 100},
 'tol': 0.0001}Optimization terminated successfully.
         Current function value: -15.158347
         Iterations: 71
         Function evaluations: 130
 final_simplex: (array([[-3.87366099, -1.66449926, -2.56251311],
       [-3.87366099, -1.66456835, -2.56251583],
       [-3.8736129 , -1.66454853, -2.56246558],
       [-3.87374307, -1.66455336, -2.56249475]]), array([-15.15834699, -15.15834699, -
↪15.15834699, -15.15834699]))
           fun: -15.158346988389001
       message: 'Optimization terminated successfully.'
          nfev: 130
           nit: 71
        status: 0
       success: True
             x: array([-3.87366099, -1.66449926, -2.56251311])Elapsed time :
↪      47.167 sec
```

[16]:
```
bso.get_basis()
```

[16]:
```
{'Be': <BasisSet(
        name                 = None
        element              = None
        family               = None
        kind                 = None

 ================s shell=================
 Contracted:
     1     2940.0000000000     0.00068000    -0.00012300
     2      441.2000000000     0.00523600    -0.00096600
     3      100.5000000000     0.02660600    -0.00483100
     4       28.4300000000     0.09999300    -0.01931400
     5        9.1690000000     0.26970200    -0.05328000
     6        3.1960000000     0.45146900    -0.12072300
     7        1.1590000000     0.29507400    -0.13343500
     8        0.1811000000     0.01258700     0.53076700
     9        0.0589000000    -0.00375600     0.58011700
 Uncontracted:
    10        0.0207821468     1.00000000
    11        0.0589000000     1.00000000

 ================p shell=================
 Contracted:
     1        3.6190000000     0.02911100
     2        0.7110000000     0.16936500
     3        0.1951000000     0.51345800
     4        0.0601800000     0.47933800
 Uncontracted:
     5        0.1892854161     1.00000000
     6        0.0601800000     1.00000000

 ================d shell=================
 Uncontracted:
     1        0.0771107088     1.00000000
     2        0.2354000000     1.00000000
 )>, 'H': <BasisSet(
        name                 = cc-pvdz
        element              = H
        family               = None
        kind                 = None

 ================s shell=================
 Contracted:
     1       13.0100000000     0.01968500
     2        1.9620000000     0.13797700
     3        0.4446000000     0.47814800
 Uncontracted:
     4        0.1220000000     1.00000000
     5        0.0297400000     1.00000000

 ================p shell=================
 Uncontracted:
     1        0.7270000000     1.00000000
     2        0.1410000000     1.00000000
 )>}
```

[17]:
```
bso.result
```

[17]:
```
 final_simplex: (array([[-3.87366099, -1.66449926, -2.56251311],
       [-3.87366099, -1.66456835, -2.56251583],
       [-3.8736129 , -1.66454853, -2.56246558],
       [-3.87374307, -1.66455336, -2.56249475]]), array([-15.15834699, -15.15834699, -
↪15.15834699, -15.15834699]))
           fun: -15.158346988389001
       message: 'Optimization terminated successfully.'
          nfev: 130
           nit: 71
        status: 0
       success: True
             x: array([-3.87366099, -1.66449926, -2.56251311])
```

Optimization of tight functions for cc-pvdz Be

[18]:
```
bsd = BasisSet.from_str(bsstr, fmt='molpro', name='cc-pvdz')
pvdzbe = {k:v for k, v in bsd.items() if k=='Be'}
```

[19]:
```
tightfs = {'Be' : [('s', 'exp', 1, (1.8,)), ('p', 'exp', 1, (4.2,))]}
```

[20]:
```
tighttmp = '''***,be-core
memory,100,m                            !allocate 500 MW dynamic memory

%geometry

%basis

%core

{rhf; wf,4,1,0}
cisd
'''
```

[21]:
```
bso = BSOptimizer(objective='cisd total energy', template=tighttmp, code=mp, mol=be,
                  fsopt=tightfs, staticbs=pvdzbe, runcore=True,
                  core=[[1,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0]], verbose=False)
```

[22]:
```
bso.run()
```

```
Script name : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages/
↪ipykernel_launcher.py
Workdir     : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages
Start time  : 2017-11-27 15:02:52.166124
================================================================================
============================STARTING OPTIMIZATION===============================
================================================================================

====================================CODE========================================
<Molpro(
        name=Molpro,
        molpropath=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin,
        executable=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin/molpro,
        scratch=/home/lmentel/scratch,
        runopts=['-s', '-n', '1', '-d', '.'],
```

(continues on next page)

(continued from previous page)

```
)>
====================================MOL=======================================
Name: Be          Charge: 0          Multiplicty: 1          Electrons: 4
Atoms:
Element    Nuclear Charge              x              y              z
Be                   4.00              0.00000        0.00000        0.00000
====================================OPTALG====================================
{'jacob': None,
 'method': 'Nelder-Mead',
 'options': {'disp': True, 'maxiter': 100},
 'tol': 0.0001}Optimization terminated successfully.
         Current function value: -0.031657
         Iterations: 29
         Function evaluations: 57
 final_simplex: (array([[ 0.62047435,  1.81858876],
       [ 0.62049921,  1.81859912],
       [ 0.62051376,  1.81852429]]), array([-0.0316572, -0.0316572, -0.0316572]))
           fun: -0.031657195978000985
       message: 'Optimization terminated successfully.'
          nfev: 57
           nit: 29
        status: 0
       success: True
             x: array([ 0.62047435,  1.81858876])Elapsed time :               20.818␣
↪sec
```

[23]:
```
bso.result
```

[23]:
```
 final_simplex: (array([[ 0.62047435,  1.81858876],
       [ 0.62049921,  1.81859912],
       [ 0.62051376,  1.81852429]]), array([-0.0316572, -0.0316572, -0.0316572]))
           fun: -0.031657195978000985
       message: 'Optimization terminated successfully.'
          nfev: 57
           nit: 29
        status: 0
       success: True
             x: array([ 0.62047435,  1.81858876])
```

Optimization of mid-bond function exponents

[24]:
```
be2X = Molecule(name="Be2", atoms=[('Be', (0.0, 0.0, -1.5)),
                                   ('H', (0.0, 0.0,  0.0), True),
                                   ('Be', (0.0, 0.0,  1.5))], sym="dnh 2", charge=0,␣
↪multiplicity=1)
```

[25]:
```
mbfs = {'H' : [('s', 'et', 4, (0.05, 2.0)), ('p', 'et', 4, (0.04, 2.0))]}
```

[26]:
```
mbtmp = '''***,h2o test
memory,100,m                              !allocate 500 MW dynamic memory

%geometry

%basis
```

(continues on next page)

(continued from previous page)

```
dummy, H

%core

{rhf; wf,8,1,0}
cisd

'''
```

[27]:
```
bso = BSOptimizer(objective='cisd total energy', template=mbtmp, code=mp, mol=be2X,
                  fsopt=mbfs, staticbs=pvdzbe, core=[2,0,0,0,0,0,0,0], verbose=False)
```

[28]:
```
bso.run()
```

```
Script name : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages/
↪ipykernel_launcher.py
Workdir     : /home/lmentel/anaconda3/envs/chemtools/lib/python3.6/site-packages
Start time  : 2017-11-27 15:03:13.366257
================================================================================
============================STARTING OPTIMIZATION===============================
================================================================================

====================================CODE========================================
<Molpro(
        name=Molpro,
        molpropath=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin,
        executable=/home/lmentel/Programs/molprop_2012_1_Linux_x86_64_i8/bin/molpro,
        scratch=/home/lmentel/scratch,
        runopts=['-s', '-n', '1', '-d', '.'],
)>
====================================MOL=========================================
Name: Be2        Charge: 0          Multiplicty: 1          Electrons: 8
Atoms:
Element    Nuclear Charge              x              y              z
Be                   4.00              0.00000        0.00000       -1.50000
H                    0.00              0.00000        0.00000        0.00000
Be                   4.00              0.00000        0.00000        1.50000
====================================OPTALG======================================
{'jacob': None,
 'method': 'Nelder-Mead',
 'options': {'disp': True, 'maxiter': 100},
 'tol': 0.0001}Warning: Maximum number of iterations has been exceeded.
 final_simplex: (array([[ 0.05114879,  1.70176733,  0.05504829,  1.9460615 ],
       [ 0.05114531,  1.70019891,  0.05505673,  1.94637608],
       [ 0.05111266,  1.70488967,  0.05504958,  1.94627325],
       [ 0.05116421,  1.7018318 ,  0.05507583,  1.94578227],
       [ 0.05111091,  1.70251424,  0.05509779,  1.94542874]]), array([-29.16470765, -
↪29.16470765, -29.16470765, -29.16470765, -29.16470765]))
           fun: -29.164707647050999
       message: 'Maximum number of iterations has been exceeded.'
          nfev: 168
           nit: 100
        status: 2
       success: False
             x: array([ 0.05114879,  1.70176733,  0.05504829,  1.9460615 ])Elapsed␣
↪time :              65.526 sec
```

[29]:
```
bso.result
```

[29]:
```
 final_simplex: (array([[ 0.05114879,  1.70176733,  0.05504829,  1.9460615 ],
       [ 0.05114531,  1.70019891,  0.05505673,  1.94637608],
       [ 0.05111266,  1.70488967,  0.05504958,  1.94627325],
       [ 0.05116421,  1.7018318 ,  0.05507583,  1.94578227],
       [ 0.05111091,  1.70251424,  0.05509779,  1.94542874]]), array([-29.16470765, -
↪29.16470765, -29.16470765, -29.16470765, -29.16470765]))
           fun: -29.164707647050999
       message: 'Maximum number of iterations has been exceeded.'
          nfev: 168
           nit: 100
        status: 2
       success: False
             x: array([ 0.05114879,  1.70176733,  0.05504829,  1.9460615 ])
```

[30]:
```
%version_information chemtools
```

[30]:

Software	Version
Python	3.6.3 64bit [GCC 7.2.0]
IPython	6.2.1
OS	Linux 4.9.0 4 amd64 x86_64 with debian 9.1
chemtools	0.8.4
Mon Nov 27 15:04:18 2017 CET	

1.2.3 Molpro tutorial

The `molpro` module contains the `Molpro` convenience class, which is python wrapper for running the Molpro program with methods for writing input files, running the calculations and parsing the output files.

[3]:
```
from chemtools.calculators.molpro import Molpro
```

Molpro wrapper

`Molpro` object can be created by either providing the path to the molpro executable with `executable` keyword argument or name of the environmental variable holding the path through `exevar`.

[5]:
```
molpro = Molpro(exevar="MOLPRO_EXE",
                runopts=["-s", "-n", "1"],
                scratch="/home/lmentel/scratch")
```

Now we can write some sample molpro input and run in from python using the `Molpro` object created earlier.

[6]:
```
inpstr = '''***,H2O finite field calculations
r=1.85,theta=104                !set geometry parameters
geometry={O;                    !z-matrix input
H1,O,r;
H2,O,r,H1,theta}

basis=avtz                      !define default basis

hf
ccsd(t)
```

(continues on next page)

(continued from previous page)

```
'''

with open('h2o.inp', 'w') as finp:
    finp.write(inpstr)
```

`run` method runs the job and returns the ouput file name

[10]:
```
output = molpro.run('h2o.inp')
```

[11]:
```
output
```

[11]:
```
'h2o.out'
```

Parsing the results

To check if the calculation finished without errors you can use `accomplished` method

[13]:
```
molpro.accomplished(output)
```

[13]:
```
True
```

[15]:
```
molpro.parse(output, 'hf total energy')
```

[15]:
```
-76.058288041466
```

[16]:
```
molpro.parse(output, 'mp2 total energy')
```

[18]:
```
molpro.parse(output, 'ccsd(t) total energy')
```

[18]:
```
-76.341780640047
```

[17]:
```
molpro.parse(output, 'regexp', regularexp=r'NUCLEAR REPULSION ENERGY\s*(-?\d+\.\d+)')
```

[17]:
```
8.99162654
```

[21]:
```
%version_information chemtools
```

[21]:

Software	Version
Python	3.6.3 64bit [GCC 7.2.0]
IPython	6.2.1
OS	Linux 4.9.0 4 amd64 x86_64 with debian 9.1
chemtools	0.8.4
Mon Nov 27 16:49:17 2017 CET	

1.2.4 Gamess(US) tutorial

The `gamessus` module contains the `GamessUS` convenience class, which is python wrapper for running the Gamess(US) program with methods for writing input files, running the calculations and parsing the output files.

[1]:
```
from chemtools.calculators.gamessus import GamessUS, GamessLogParser
```

GamessUS wrapper

Now we instantiate the object by calling `Gamess` with arguments corresponding to the local installation of GAMESS(US)

[5]:
```
gamess = GamessUS(exevar="GAMESS_EXE",
                  version="00",
                  runopts=["1"],
                  scratch="/home/lmentel/scratch")
```

[6]:
```
gamess.rungms
```

[6]:
```
'/home/lmentel/Programs/gamess-us-aug2016/rungms'
```

[7]:
```
inpstr = """ $CONTRL scftyp=rhf runtyp=energy maxit=30 mult=1 ispher=1
     itol=30 icut=30 units=bohr cityp=guga qmttol=1.0e-8 $END
 $SYSTEM timlim=525600 mwords=100 $END
 $SCF dirscf=.false. $END
 $CIDRT iexcit=2 nfzc=0 ndoc=1 nval=27 group=d2h stsym=ag
        mxnint=14307305 $END
 $GUGDIA prttol=1.0e-6 cvgtol=1.0e-10 $END
 $DATA
H2 cc-pVTZ
dnh 2

H    1.00       0.000000       0.000000       0.700000
S   3
  1     33.8700000              0.0060680
  2      5.0950000              0.0453080
  3      1.1590000              0.2028220
S   1
  1      0.3258000              1.0000000
S   1
  1      0.1027000              1.0000000
P   1
  1      1.4070000              1.0000000
P   1
  1      0.3880000              1.0000000
D   1
  1      1.0570000              1.0000000

 $END
"""
```

Write the input file

[8]:
```
inpfile = "h2_eq_pvtz_fci.inp"
with open(inpfile, 'w') as inp:
    inp.write(inpstr)
```

Run the calculation

[12]:
```
logfile = gamess.run(inpfile)
```

[13]:
```
logfile
```

[13]:
```
'h2_eq_pvtz_fci.log'
```

Parsing the results

`GamessUS` has only rudimentary parsing methods implemented for the purpouses of being compliant with basis set optimizer API, however there is a dedicated parser class implemented in `chemtools` called `GamessLogParser`. There is also a separate class wrapper to parsing and writing Gamess(US) input files, and another one for reading binary files produced during the calculation such as integral files and the dictionary file.

[15]:
```
gamess.accomplished(logfile)
```
[15]:
```
True
```

[14]:
```
gamess.parse(logfile, 'hf total energy')
```
[14]:
```
-1.1329605255
```

[16]:
```
gamess.parse(logfile, "cisd total energy")
```
[16]:
```
-1.1723345936
```

[18]:
```
gamess.parse(logfile, "correlation energy")
```
[18]:
```
-0.039374068100000104
```

[17]:
```
gamess.parse(logfile, 'regexp', r'NUMBER OF CONFIGURATIONS\s*=\s*(\d+)')
```
[17]:
```
82.0
```

[19]:
```
%version_information chemtools
```
[19]:

Software	Version
Python	3.6.3 64bit [GCC 7.2.0]
IPython	6.2.1
OS	Linux 4.9.0 4 amd64 x86_64 with debian 9.1
chemtools	0.8.4
Mon Nov 27 19:27:43 2017 CET	

1.2.5 Jupyter Notebooks

The tutorials are also avaiable as Jupyter notebooks:

- BasisSet
- Basis set optimization
- Gamess(US) wrapper
- Molpro wrapper

1.3 API Reference

1.3.1 Calculators

Base Calculator

class Calculator(*exevar=None*, *executable=None*, *runopts=None*, *scratch=None*)
Abstract class that should be subclassed when adding a new code interace.

accomplished(*fname*)
Return True if the job completed without errors

executable
Set the executable

parse(*fname*, *objective*, *regexp=None*)
Parse the value of the `objective` from the file `fname`

run(**args*)
run a single job

run_multiple(**args*)
Run multiple jobs

scratch
Return scratch

write_input()
write the input file in the format of the code used

Dalton

class Dalton(*name='Dalton'*, ***kwargs*)
Wrapper for running Dalton program

parse(*fname*, *objective*, *regularexp=None*)
Parse a value from the output file `fname` based on the `objective`.

If the value of the `objective` is `regexp` then the `regularexp` will be used to parse the file.

run(*fname*)
Run a single job

Args:

fname [dict] A dictionary with keys `mol` and `dal` and their respective file name strings as values

Returns:

out [str] Name of the dalton output file

run_multiple(*fnames*)
Spawn two single jobs as paralell processes

write_input(*fname*, *template*, *basis*, *mol*, *core*)
Write dalton input files: `fname.dal` and `system.mol`

Args:

fname [str] Name of the input file `.dal`

template [dict] Dictionary with templates for the `dal` and `mol` with those strings as keys and actual templates as values

basis [dict] An instance of *BasisSet* class or a dictionary of *BasisSet* objects with element symbols as keys

mol [*chemtools.molecule.Molecule*] Molecule object with the system geometry

core [str] Core definition

DMFT

class Dmft(*logfile*)
dmft class

parse_gamess()
Parse gamess-us log file to get the neccessary data to write dmft input and set defaults.

run()
Run a single dmft job.

write_input(*functional=None*, *a1=None*, *b1=None*, *a2=None*, *b2=None*, *printlevel=None*, *analyze=None*)
Write dmft input based on information in the gamess log file.

GAMESS-US

class GamessUS(*name='GamessUS'*, *version='00'*, *runopts=None*, ***kwargs*)
Container object for Gamess-us jobs.

accomplished(*outfile*)
Return True if Gamess(US) job finished without errors.

get_command(*inpfile*)
Return the command to execute

parse(*output*, *objective*, *regexp=None*)
Parser GAMESS(US) output file to get the objective.

remove_dat(*inpfile*)
Remove the gamess dat file if it exists in the scratch directory.

run(*inpfile*, *logfile=None*, *remove_dat=True*)
Run a single gamess job interactively - without submitting to the queue.

run_multiple(*inputs*)
Run multiple jobs

runopts
Return the runopts

version
Return the version.

write_input(*fname*, *template=None*, *mol=None*, *basis=None*, *core=None*)
Write the molpro input to "fname" file based on the information from the keyword arguments.

Args:

mol [*chemtools.molecule.Molecule*] Molecule object instance

basis [dict or *BasisSet*] An instance of *BasisSet* class or a dictionary of *BasisSet* objects with element symbols as keys

core [list of ints] Molpro core specification

template [str] Template of the input file

fname [str] Name of the input file to be used

Molpro

class Molpro(*name='Molpro'*, ***kwargs*)
Wrapper for the Molpro program.

accomplished(*fname*)
Return True if the job completed without errors

parse(*fname*, *objective*, *regularexp=None*)
Parse a value from the output file `fname` based on the `objective`.

If the value of the `objective` is `regexp` then the `regularexp` will be used to parse the file.

run(*inpfile*)
Run a single molpro job interactively - without submitting to the queue.

run_multiple(*inputs*)
Run a single molpro job interactively - without submitting to the queue.

write_input(*fname=None*, *template=None*, *mol=None*, *basis=None*, *core=None*)
Write the molpro input to "fname" file based on the information from the keyword arguments.

Args:

mol [*chemtools.molecule.Molecule*] Molecule object instance

basis [dict or *BasisSet*] An instance of *BasisSet* class or a dictionary of *BasisSet* objects with element symbols as keys

core [list of ints] Molpro core specification

template [`str`] Template of the input file

fname [`str`] Name of the input file to be used

PSI4

class Psi4(*name='Psi4'*, ***kwargs*)
Wrapper for the Psi4 program.

parse(*fname*, *objective*, *regularexp=None*)
Parse a value from the output file `fname` based on the `objective`.

If the value of the `objective` is `regexp` then the `regularexp` will be used to parse the file.

run(*inpfile*)
Run a single Psi4 job interactively - without submitting to the queue.

run_multiple(*inputs*)
Run a single Psi4 job interactively - without submitting to the queue.

write_input(*fname*, *template*, *mol=None*, *basis=None*, *core=None*)
Write the Psi4 input to "fname" file based on the information from the keyword arguments.

Args:

mol [*chemtools.molecule.Molecule*] Molecule object instance

basis [dict or *BasisSet*] An instance of *BasisSet* class or a dictionary of *BasisSet* objects with element symbols as keys

core [list of ints] Psi4 core specification

template [`str`] Template of the input file

fname [str] Name of the input file to be used

1.3.2 Basis Set Tools

BasisSet module

class BasisSet(*name*, *element*, *family=None*, *kind=None*, *functions=None*, *info=None*)
Basis set module supporting basic operation on basis sets.

Args:

name [str] Name of the basis set

element [str] Symbol of the element

Kwargs:

kind [str] Classification of the basis set functions, *diffuse*, *tight*

family [str] basis set family

functions [dict] Dict of functions with $s, p, d, f, \ldots$ as keys

params [list of dicts] Parameters for generating the functions according to the model

append(*other*)
Append functions from another BasisSet object

Args:

other [BasisSet] BasisSet object whose functions will be added to the existing ones

completeness_profile(*zetas*)
Calculate the completeness profile of each shell of the basis set

Args:

zetas [numpy.array] Scaning exponents

Returns:

out [numpy.array] numpy array with values of the profile (shells, zetas)

contraction_matrix(*shell*)
Return the contraction coefficients for a given shell in a matrix form with size *ne* * *nc*, where *ne* is the number of exponents and *nc* is the number of contracted functions

Args:

shell [str] shell label, $s, p, d, \ldots$

Returns:

out [2D numpy.array] 2D array with contraction coefficients

contraction_scheme()
Return a string describing the contraction scheme.

contraction_type()
Try to determine the contraction type: segmented, general, uncontracted, unknown.

contractions_per_shell()
Calculate how many contracted functions are in each shell.

Returns: out : list of ints

classmethod from_file(*fname=None*, *fmt=None*, *name=None*)
Read and parse a basis set from file and return a BasisSet object

Args:

fname [str] File name

fmt [str] Format of the basis set in the file (*molpro*, *gamessus*)

name [str] Name of the basis set

Returns:

out [BasisSet or dict] Basisset object parsed from file or dictionary of BasisSet objects

classmethod from_json(*jsonstring*, ***kwargs*)
Instantiate the *BasisSet* object from a JSON string

Args:

jsonstring: str A JSON serialized string with the basis set

classmethod from_optpars(*x0*, *funs=None*, *name=None*, *element=None*, *explogs=False*)
Return a basis set object generated from a sequence based on the specified arguments.

Args:

x0 [list or numpy.array] Parameters to generate the basis set as a continuous list or array

funs [list of tuples] A list of tuple specifying the shell type, number of functions and parameters, e.g. *[('s', 'et', 4, (0.5, 2.0)), ('p', 'et', 3, (1.0, 3.0))]*

name [str] Name of the basis set

element [str] Chemical symbol of the element

Returns: out : BasisSet

static from_pickle(*fname*, ***kwargs*)
Read a pickled BasisSet object from a pickle file

Args:

fname [str] File name containing the BasisSet

kwargs [dict] Extra arguments for the *pickle.load* method

Raises:

UnicodeDecodeError When you try to read a python2 pickle using python3, to fix that use *encoding='latin1'* option

classmethod from_sequence(*funs=None*, *name=None*, *element=None*)
Return a basis set object generated from a sequence based on the specified arguments.

Args:

funs [list of tuples] A list of tuple specifying the shell type, number of functions and parameters, e.g. *[('s', 'et', 4, (0.5, 2.0)), ('p', 'et', 3, (1.0, 3.0))]*

name [str] Name of the basis set

element [str] Chemical symbol of the element

Returns: out : BasisSet

classmethod from_str(*string*, *fmt=None*, *name=None*)
Parse a basis set from string

Args:

string [str] A string with the basis set

fmt [str] Format of the basis set in the file: *molpro*, *gamessus*

name [str] Name of the basis set

Returns:

out [BasisSet or dict] Basisset object parsed from string or dictionary of BasisSet objects

get_exponents(*asdict=True*)
Return the exponents of a given shell or if the shell isn't specified return all of the available exponents

Args:

asdict (bool): if *True* a dict with exps per shell is returned

nf(*spherical=True*)
Calculate the number of basis functions

Args:

spherical [bool] flag indicating if spherical or cartesian functions should be used, default: True

Returns:

res [int] number of basis functions

normalization()
For each function (contracted) calculate the norm and return a list of tuples containing the shell, function index and the norm respectively.

normalize()
Normalize contraction coefficients for each contracted functions based on the primitive overlaps so that the norm is equal to 1.

nprimitive(*spherical=True*)
Return the number of primitive functions assuming sphrical or cartesian gaussians.

Args:

spherical [bool] A flag to select either spherical or cartesian gaussians

Returns:

out [int] Number of primitive function in the basis set

partial_wave_expand()
From a given basis set with shells spdf... return a list of basis sets that are subsets of the entered basis set with increasing angular momentum functions included [s, sp, spd, spdf, ...]

primitives_per_contraction()
Calculate how many primities are used in each contracted function.

Returns: out : list of ints

primitives_per_shell()
Calculate how many primitive functions are in each shell.

Returns: out : list of ints

print_functions(*efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set.

Args:

efmt [str] string describing output format for the exponents, default: "20.10f"

cfmt [str] string describing output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string

shell_overlap(*shell*)
Calculate the overlap integrals for a given shell

Args:

shell [str] Shell

Returns:

out [numpy.array] Overlap integral matrix

sort(*reverse=False*)
Sort shells in the order of increasing angular momentum and for each shell sort the exponents.

Args:

reverse [bool] If *False* sort the exponents in each shell in the descending order (default), else sort exponents in ascending order

to_cfour(*comment=''*, *efmt='15.8f'*, *cfmt='15.8f'*)
Return a string with the basis set in (new) CFOUR format.

Args:

comment [str] comment string

efmt [str] string describing output format for the exponents, default: "20.10f"

cfmt [str] string describing output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string in Cfour format

to_dalton(*fmt='prec'*)
Return a string with the basis set in DALTON format.

Args:

fmt [str]

string describing output format for the exponents and coefficents

- *prec* "20.10f"
- *default* "10.4f"
- or python format string e.g. "15.8f"

Returns:

res [str] basis set string in Dalton format

to_gamessus(*efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set in GAMESS(US) format.

Args:

efmt [str] string describing output format for the exponents, default: "20.10f"

cfmt [str] string describing output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string in Gamess(US) format

to_gaussian(*efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set in Gaussian format.

Args:

efmt [str] string describing output format for the exponents, default: "20.10f"

cfmt [str] string describing output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string in Gaussian format

to_json(*fname=None*, ***kwargs*)
Serizalize the basis set object to JSON format

to_latex(*efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set as LaTeX table/

Args:

efmt [str] Output format for the exponents, default: "20.10f"

cfmt [str] Output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string in LaTeX format

to_molpro(*withpars=False*, *efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set in MOLPRO format.

Args:

withpars [bool] A flag to indicate whether to wrap the basis with *basis={ }* string

efmt [str] Output format for the exponents, default: "20.10f"

cfmt [str] Output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string

to_nwchem(*efmt='20.10f'*, *cfmt='15.8f'*)
Return a string with the basis set in NWChem format.

Args:

efmt [str] string describing output format for the exponents, default: "20.10f"

cfmt [str] string describing output format for the contraction coefficients, default: "15.8f"

Returns:

res [str] basis set string in NwChem format

to_pickle(*fname=None*)
Save the basis set in pickle format under the filename *fname*

Args:

fname [str] File name

uncontract (*copy=False*)
Uncontract the basis set. This replaces the contraction coefficients in the current object.

Args:

copy [bool] If *True* return an uncontracted copy of the basis set rather than uncontracting in place, default is *False*.

has_consecutive_indices (*shell*)
Check if all the contracted functions have consecutive indices

Args:

shell [dict] Basis functions for a given shell asa dict with structure `{'e' : np.array(), 'cf': [np.array(), np.array(), ...]}`

reorder_shell_to_consecutive (*shell*)
Reorder the exponents so that the indices of the contracted functions have consecutive inidices.

Args:

shell [dict] Basis functions for a given shell asa dict with structure `{'e' : np.array(), 'cf': [np.array(), np.array(), ...]}`

Returns:

shell [dict] Same shell as on input but with reordered exponents and relabelled contracted functions

merge (*first*, *other*)
Merge functions from two BasisSet objects

Args:

first [BasisSet] First *BasisSet* object to merge

other [BasisSet] Second *BasisSet* object to merge

Returns:

our [BasisSet] BasisSet instance with functions from *first* and *other* merged

primitive_overlap (l, a, b)
Calculate the overlap integrals for a given shell *l* and two sets of exponents

$$S(\zeta_i, \zeta_j) = \frac{2^{l+\frac{3}{2}} \left(\zeta_1 \zeta_2\right)^{\frac{l}{2}+\frac{3}{4}}}{\left(\zeta_1 + \zeta_2\right)^{l+\frac{3}{2}}}$$

Args:

l [int] Angular momentum quantum number of the shell

a (M,) [numpy.array] First vector of exponents

b (N,) [numpy.array] Second vector of exponents

Returns:

out (M, N) [numpy.array] Overlap integrals

nspherical (l)
Calculate the number of spherical components of a function with a given angular momentum value *l*.

ncartesian (l)
Calculate the number of cartesian components of a function with a given angular momentum value *l*.

zetas2legendre(*zetas*, *kmax*)

From a set of exponents (*zetas*), using least square fit calculate the expansion coefficients into the legendre polynomials of the order *kmax*.

Args:

kmax [int] length of the legendre expansion

zetas [list] list of exponents (floats) to be fitted

Returns:

coeff [numpy.array] numpy array of legendre expansion coeffcients of length kmax

zetas2eventemp(*zetas*)

From a set of exponents (*zetas*), using least square fit calculate the approximate $alpha$ and $beta$ parameters from the even tempered expansion.

Args:

zetas [list] list of exponents (floats) to be fitted

Returns:

coeff [numpy.array] numpy array of legendre expansion coeffcients of length kmax

eventemp(*numexp*, *params*)

Generate a sequence of nf even tempered exponents accodring to the even tempered formula

$$\zeta_i = \alpha \cdot \beta^{i-1}$$

Args:

numexp [int] Number fo exponents to generate

params [tuple of floats] Alpha and beta parameters

Returns:

res [numpy array] Array of generated exponents (floats)

welltemp(*numexp*, *params*)

Generate a sequence of nf well tempered exponents accodring to the well tempered fromula

$$\zeta_i = \alpha \cdot \beta^{i-1} \cdot \left[1 + \gamma \cdot \left(\frac{i}{N}\right)^{\delta}\right]$$

Args:

numexp [int] Number fo exponents to generate

params [tuple of floats] Alpha, beta, gamma and delta parameters

Returns:

res [numpy.array] Array of generated exponents (floats)

legendre(*numexp*, *coeffs*)

Generate a sequence of nf exponents from expansion in the orthonormal legendre polynomials as described in: Peterson, G. A. et.al J. Chem. Phys., Vol. 118, No. 3 (2003), eq. (7).

$$\ln \zeta_i = \sum_{k=0}^{k_{\max}} A_k P_k \left(\frac{2j-2}{N-1} - 1\right)$$

Args:

numexp [int] Number fo exponents to generate

params [tuple of floats] Polynomial coefficients (expansion parameters)

Returns:

res [numpy array] Array of generated exponents (floats)

xyzlist (*l*)

Generate an array of l_x, l_y, l_z components of cartesian gaussian with a given angular momentum value in canonical order.

For exampe:

- `l = 0` generates the result `array([[0, 0, 0]])`
- `l = 1` generates `array([[1, 0, 0], [0, 1, 0], [0, 0, 1])`
- etc.

The functions are coded by triples of powers of *x*, *y*, *z*, namely `[1, 2, 3]` corresponds to xy^2z^3.

Args:

l [int] Angular momentum value

Returns:

out ((l+1)*(l+2)/2, 3) [numpy.array] Array of monomial powers, where columns correspond to *x*, *y*, and *z* respectively and rows correspond to functions.

zlmtoxyz (*l*)

Generates the expansion coefficients of the real spherical harmonics in terms of products of cartesian components. Method based on [Schelegel1995]

Args:

l [int] Angular momentum value

Returns:

out $((l+1)*(l+2)/2, 2*l+1)$ [numpy.array] Expansion coefficients of real spherical harmonics in terms of cartesian gaussians

basisparse module

Parsing basis set from different formats.

class NumpyEncoder (*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

default (*obj*)

If input object is an ndarray it will be converted into a dict holding the data and dtype.

get_l (*shell*)

Return the orbital angular momentum quantum number for a given subshell

merge_exponents (*a*, *b*)

Concatenate the arrays *a* and *b* using only the unique items from both arrays

Args: a : numpy.array b : numpy.array

Returns:

res [3-tuple of numpy arrays]

- res[0] sorted union of *a* and *b*
- res[1] indices of *a* items in res[0]
- res[2] indices of *b* items in res[0]

parse_basis(*string*, *fmt=None*)
A wrapper for parsing the basis sets in different formats.

Args:

string [str] A string with the basis set

fmt [str] Format in which the basis set is specified

Returns:

out [dict] A dictionary of parsed basis sets with element symbols as keys and basis set functions as values

parse_gamessus_basis(*string*)
Parse the basis set into a list of dictionaries from a string in gamess format.

parse_gamessus_function(*lines*)
Parse a basis set function information from list of strings into three lists containg: exponents, indices, coefficients.

Remeber that python doesn't recognise the *1.0d-3* format where *d* or *D* is used to the regex subsitution has to take care of that.

parse_gaussian_basis(*string*)
Parse the basis set into a list of dictionaries from a string in gaussian format.

parse_gaussian_function(*lines*)
Parse a basis set function information from list of strings into three lists containg: exponents, indices, coefficients.

Remeber that python doesn't recognise the *1.0d-3* format where *d* or *D* is used to the regex subsitution has to take care of that.

parse_molpro_basis(*string*)
Parse basis set from a string in Molpro format.

parse_molpro_shell(*expsline*, *coeffs*)
Parse functions of one shell in molpro format.

basisopt module

Optimization of basis set exponents and contraction coefficients.

`basisopt` is a module containing flexible methods for optimizing primitive exponents of basis sets

class BSOptimizer(*objective=None*, *core=None*, *template=None*, *regexp=None*, *verbose=False*, *code=None*, *optalg=None*, *mol=None*, *fsopt=None*, *staticbs=None*, *fname=None*, *uselogs=True*, *runcore=False*, *penalize=None*, *penaltykwargs=None*, *logfile=None*)
Basis Set Optimizer class is a convenient wrapper for optimizing primitive exponents of Gaussian basis sets using different code interfaces for performing the actual electronic structure calculations.

Args:

objective [str or callable] Name of the objective using which the optimization will be evaluated, if it is a callable/function it should take the output name as argument

code [*Calculator*] Subclass of the *Calculator* specifying the electronic structure program to use

mol [:py:class:‘ Molecule <chemtools.molecule.Molecule>‘] Molecule specifying the system

staticbs [dict or *BasisSet*] Basis set or `dict` of basis set with basis sets whose exponents are not going to be optimized

fsopt [dict] A dictionary specifying the basis set to be optimized, the keys should be element/atom symbol and the values should contain lists of 4-tuples composed of: shell, type, number of functions and parameters/exponents, for example,

```
>>> fs = {'H' : [('s', 'et', 10, (0.5, 2.0))]}
```

describes 10 `s`-type exponents for H generated from even tempered formula with parameters 0.5 and 2.0

optalg [dict] A dictionary specifying the optimization algorithm and its options

fname [str] Name of the job/input file for the single point calculator

uselogs [bool] Use natural logarithms of exponents in optimization rather than their values, this option is only relevant if functions asre given as `exp` or `exponents`

regexp [str] Regular expression to use in search for the objective if `objective` is regexp

runcore [bool] Flag to mark wheather to run separate single point jobs with different numbers of frozen core orbitals to calculate core energy

penalize [bool] Flag enabling the use of penalty function, when two exponent in any shell are too close the objective is multiplied by a penalty factor calculated in *get_penalty*

penaltykwargs [dict] Keyword arguments for the penalty function, default `{'alpha' : 25.0, 'smallestonly' : True}`

get_basis(*name=None*, *element=None*)

Construct the BasisSet object from the result of the optimized exponents and function definition.

Args:

name [str] Name to be assigned to the basis set

element [str] Element symbol for the basis set

Returns:

basis [chemtools.basisset.BasisSet] *BasisSet* object with the optimized functions

get_x0()

Collect all the parameters in an array of consecutive elements.

header()

Return the basic information about the execution environment and optimization settings.

jobinfo()

Return the information on the optimization objects and parameters

run()

Start the basis set optimization

Returns:

res [OptimizeResult] An instance of the `scipy.optimize.OptimizeResult` class

get_basis_dict(*bso*, *x0*)

Return a dictionary with `BasisSet` objects as values and element symbols as keys. The dictionary is composed based on the current parameters `x0` and attributes of the *BSOptimizer* including the `staticbs`

get_penalty(*bsdict*, *alpha=25.0*, *smallestonly=True*)

For a given dict of basis sets calculate the penalty for pairs of exponents being too close together.

Args:

bsdict [dict] Dictionary of `BasisSet` objects

alpha [float] Parameter controlling the magnitude and range of the penalty

smallestonly [bool] A flag to mark whether to use only the smallest ratio to calculate the penalty or all smallest ratios from each shell and calculate the penalty as a product of individual penalties

For each basis and shell within the basis ratios between pairs of sorted exponents are calculated. The minimal ratio (closest to 1.0) is taken to calculate the penalty according to the formula

$$P = 1 + \exp(-\alpha(r - 1))$$

where r is the ratio between the two closest exponents (>1).

opt_multishell(*shells=None*, *nfps=None*, *guesses=None*, *max_params=5*, *opt_tol=0.0001*, *save=False*, *bsopt=None*, ***kwargs*)

Optimize a basis set by saturating the function space shell by shell

Kwargs:

shells (list of strings): list of shells to be optimized, in the order the optimization should be performed,

nfps (list of lists of integers): list specifying a set of function numbers to be scanned per each shell,

guesses (list of lists of floats): list specifying a set of starting parameters per each shell,

max_params (int) maximal number of parameters to be used in the legendre expansion, (length of the expansion)

opt_tol (float): threshold controlling the termination of the shell optimization, if energy difference between two basis sets with subsequent number of functionsis larger than this threshold, another function is added to this shell and parameters are reoptimized

kwargs: options for the basis set optimization driver, see driver function from the basisopt module

opt_shell_by_nf(*shell=None*, *nfs=None*, *max_params=5*, *opt_tol=0.0001*, *save=False*, *bsopt=None*, ***kwargs*)

For a given shell optimize the functions until the convergence criterion is reached the energy difference for two consecutive function numbers is less than the threshold

Kwargs:

shell [string] string label for the shell to be optimized

nfs [list of ints] list of integers representing the number of basis functions to be inceremented in the optimization,

max_params [int] maximal number of parameters to be used in the legendre expansion, (length of the expansion)

opt_tol [float] threshold controlling the termination of the shell optimization, if energy difference between two basis sets with subsequent number of functionsis larger than this threshold, another function is added to this shell and parameters are reoptimized,

save [bool] a flag to trigger saving all the optimized basis functions for each shell,

kwargs [dict] options for the basis set optimization driver, see driver function from the basisopt module

Returns: BasisSet object instance with optimized functions for the specified shell

Raises:

ValueError: when *shell* is different from *s*, *p*, *d*, *f*, *g*, *h*, *i* when number of parameters equals 1 and there are more functions when there are more parameters than functions to optimize

run_core_energy (*x0*, **args*)

Funtion for running two single point calculations and parsing the resulting energy (or property) as specified by the objective function, primarily designed to extract core energy.

Args:

x0: list or numpy.array contains a list of parameters to be optimized, may be explicit exponents or parametrized exponents in terms of some polynomial

args: tuple of dicts bsopt, bsnoopt, code, job, mol, opt, needed for writing input and parsing output

Returns: parsed result of the single point calculation as speficied by the objective function in the "job" dictionary

run_total_energy (*x0*, **args*)

Funtion for running a single point calculation and parsing the resulting energy (or property) as specified by the objective function.

Args:

x0 [list or numpy.array] contains a list of parameters to be optimized, may be explicit exponents or parametrized exponents in terms of some polynomial

args [tuple of dicts] bsopt, bsnoopt, code, job, mol, opt, needed for writing input and parsing output

Returns: parsed result of the single point calculation as speficied by the objective function in the "job" dictionary

CBS module

Complete Basis Set (CBS) extrapolation techniques.

Module for Complete Basis Set (CBS) Extrapolations.

expo ()

CBS extrapolation formula by exponential Dunning-Feller relation.

$$E^{HF}(X) = E_{CBS} + a \cdot \exp(-bX)$$

Returns: function object

exposqrt (*twopoint=True*)

Three-point formula for extrapolating the HF reference energy[2].

$$E^{HF}(X) = E_{CBS} + a \cdot \exp(-b\sqrt{X})$$

Args:

twpoint [bool] A flag marking the use of two point extrapolation with *b=9.0*

Returns: funtion object

exposum ()

Three point extrapolation through sum of exponentials expression

$$E(X) = E_{CBS} + a \cdot \exp(-(X-1)) + b \cdot \exp(-(X-1)^2)$$

[2] Karton, A., & Martin, J. M. L. (2006). Comment on: "Estimating the Hartree-Fock limit from finite basis set calculations" [Jensen F (2005) Theor Chem Acc 113:267]. Theoretical Chemistry Accounts, 115, 330–333. doi:10.1007/s00214-005-0028-6

extrapolate(*x*, *energy*, *method*, ***kwargs*)
An interface for performing CBS extrapolations using various methods.

Args:

x [numpy.array] A vector of basis set cardinal numbers

energy [numpy.array] A vector of corresponding energies

method [str] Method/formula to use to perform the extrapolation

Kwargs: Keyword arguments to be passed to the requested extrapolation function using the *method* argument

poly(*p=0.0*, *z=3.0*, *twopoint=True*)
CBS extrapolation by polynomial relation.

$$E(X) = E_{CBS} + \sum_i a_i \cdot (X + P)^{-b_i}$$

Kwargs:

twpoint [bool] A flag for choosing the two point extrapolation

z [float or list of floats] Order of the polynomial, *default=3.0*

p [float] A parameter modifying the cardinal number, *default=0.0*

uste(*method='CI'*)
CBS extrapolation using uniform singlet and triplet pair extrapolation (USTE) scheme [Varandas2007].

Args:

x [int] Cardinal number of the basis set

e_cbs [float] Approximation to the energy value at the CBS limit

a [float] Empirical A3 parameter

method [str] One of: *ci*, *cc*

Returns: function object

Molecule module

Module for handling atoms and molecules.

class Atom(*identifier*, *xyz=(0.0, 0.0, 0.0)*, *dummy=False*, *id=None*)
Basic atom class representing an atom.

move(*x=0.0*, *y=0.0*, *z=0.0*)
Move atom to a set of new coordinates given in xyz

class Molecule(*name=''*, *atoms=None*, *unique=None*, *sym=''*, *charge=0*, *multiplicity=1*)
Molecule class handling all the operations on single molecules

get_distance(*atom1*, *atom2*)
Calcualte the distance between two atoms.

nele()
Get the total number of electrons in a molecule.

unique()
Get a list of unique atom specified by unique keyword

parsetools module

Module with convenience functions for parsing files

contains (*string*, *query*)
Check if *string* contains *query*

getchunk (*filename*, *startlno*, *endlno*)
Get a list of lines from a file between specified line numbers *startlno* and *endlno*.

Args:

filename [str] Name of the file to process

startlno [int] Number of the first line to obtain

endlno [int] Number of the last line to obtain

Returns:

lines [list] A list of lines from the file *filename* between line numbers *startlno* and *endlno*

getlines (*filename*, *tolocate*)
Return the lines from the files based on *tolocate*

Args:

filename [str] Name of the file

tolocate [list of tuples] List of tuples with strings to find (queries) as first elements and integer offset values as second

Return:

locatelinenos (*filename*, *tolocate*)
Given a file and a list of strings return a dict with string as keys and line numbers in which they appear a values.

Args:

filename [str] Name of the file

tolocate [list of tuples] List of tuples with strings to find (queries) as first elements and integer offset values as second

Returns:

out [dict] Dictionary whose keys are indices corresponding to item in input list and values are lists of line numbers in which those string appear

TODO:

- add option to ignore the case of the strings to search

parsepairs (*los*, *sep='='*)
Parse a given list of strings "los" into a dictionary based on separation by "sep" character and return the dictionary.

sliceafter (*seq*, *item*, *num*)
Return "num" elements of a sequence "seq" present after the item "item".

slicebetween (*string*, *start*, *end*)
Return a slice of the *string* between phrases *start* and *end*.

take (*seq*, *num*)
Iterate over a sequence *seq num* times and return the list of the elements iterated over.

submitgamess module

main()
Script for submitting gamessus jobs to the queue.

remove_dat(*path*, *datfile*)
Remove the dat file from the ASCII scratch.

submit_ll(*args*)
Write the run script for LoadLeveller and submit it to the queue.

submit_pbs(*args*)
Write the run script for PBS and submit it to the queue.

submit_slurm(*args*)
Write the run script for SLURM and submit it to the queue.

submitmolpro module

main()
Script for submitting molpro job to the queue.

set_defaults(*args*)
Set some useful default values and add them to args

submit_pbs(*args*)
Write the run script for PBS and submit it to the queue.

gamessorbitals module

Orbitals class

class Orbitals(**args*, ***kwargs*)
A convenience class for handling GAMESS(US) orbitals.

assign_lz_values(*decimals=6*, *tolv=0.01*)
Determine the eigenvalues of the Lz operator for each nondegenerate or a combination of degenerate orbitals and assign them to *lzvals* column in the DataFrame

The Lz integrals over atomic orbitals are read from the dictionary file record No. 379.

Args:

decimals [int] Number of decimals to keep when comparing float eigenvalues

tolv [float] Threshold for keeping wights of the eiegenvalues (squared eigenvector components)

WARNING: currently this only works if the eigenvalues on input are sorted

fragment_populations()
Calculate the fragment populations and assign each orbital to a fragment

classmethod from_files(*name=None*, *logfile=None*, *dictfile=None*, *datfile=None*)
Initialize the *Orbitals* instance based on orbital information parsed from the *logfile* and read from the *dictfile*.

Args:

name [str] One of *hf* or *ci*

logfile [str] Name of the GAMESS(US) log file

dictfile [str] Name of the GAMESS(US) dictionary file .F10

datfile [str :] Name of the GAMESS(US) dat file

lzmos(*ETOLLZ=1e-06*, *TOLV=0.01*)
A python rewrite of the GAMESS(US) LZMOS subroutine (from symorb.src) for analyzing the Lz composition of the orbtials

Args: ETOLLZ : float TOLV : float

check_duplicates(*a*, *decimals=6*)
This funciton assumes that the array *a* is sorted

http://stackoverflow.com/questions/25264798/checking-for-and-indexing-non-unique-duplicate-values-in-a-numpy-array
http://stackoverflow.com/questions/5426908/find-unique-elements-of-floating-point-array-in-numpy-with-comparison-using-a-d

gamessreader module

GamessReader : reading gamess binary files.

class BinaryFile(*filename*, *mode='r'*, *order='fortran'*)
Class representing a binary file (C or Fortran ordered).

file = None
The file handler.

order = None
The order for file ('c' or 'fortran').

read(*dtype*, *shape=(1,)*)
Read an array of *dtype* and *shape* from current position.

shape must be any tuple made of integers or even () for scalars.

The current position will be updated to point to the end of read data.

seek(*offset*, *whence=0*)
Move to new file position.

Argument offset is a byte count. Optional argument whence defaults to 0 (offset from start of file, offset should be >= 0); other values are 1 (move relative to current position, positive or negative), and 2 (move relative to end of file, usually negative, although many platforms allow seeking beyond the end of a file). If the file is opened in text mode, only offsets returned by tell() are legal. Use of other offsets causes undefined behavior.

tell()
Returns current file position, an integer (may be a long integer).

write(*arr*)
Write an *arr* to current position.

The current position will be updated to point to the end of written data.

class DictionaryFile(*filename*, *irecln=4090*, *int_size=8*)
Wrapper for reading GAMESS(US) dictionary file (*.F10).

read_record(*nrec*, *dtype=None*)
Read a logical record 'rec' from the dictionary file and return a numpy array of type defined in the 'records' list, and size defined through 'self.ifilen' array.

class GamessFortranReader(*log*)

Class for holding method for reading gamess binary files: $JOB.F08 : two electron integrals over AO's, $JOB.F09 : two electron integrals over MO's, $JOB.F10 : the dictionary file with 1-e integrals, orbitals etc., $JOB.F15 : GUGA and ORMAS two-electron reduced density matrix,

get_onee_size(*aos=True*)
Get the size of the vector holding upper (or lower) triangle of a square matrix of size naos or nmos.

get_twoe_size(*aos=False*)
Get the size of the 1d vector holding upper (or lower) triangle of a supermatrix of size nmos (2RDM and two-electrons integrals).

read_rdm2(*filename=None*, *nmo=None*)
Read the 2rdm from the gamess-us file

read_twoeao(*filename=None*, *nmo=None*)
Read the two electron integrals from the gamess-us file

read_twoemo(*filename=None*, *nmo=None*)
Read the two electron integrals from the gamess-us file

class GamessReader(*log*)

Class for holding method for reading gamess binary files:

- $JOB.F08 : two electron integrals over AO's,
- $JOB.F09 : two electron integrals over MO's,
- $JOB.F15 : GUGA and ORMAS two-electron reduced density matrix,

TODO: CI coefficients, and CI hamiltonian matrix elements.

get_onee_size(*aos=True*)
Get the size of the vector holding upper (or lower) triangle of a square matrix of size naos or nmos.

get_twoe_size()
Get the size of the 1d vector holding upper (or lower) triangle of a supermatrix of size nmos (2RDM and two-electrons integrals).

read_rdm2(*filename=None*, *nmo=None*)
Read the 2rdm from the gamess-us file

read_twoeao(*filename=None*, *nmo=None*)
Read the two electron integrals from the gamess-us file

class SequentialFile(*filename*, *logfile=None*)

get_index_buffsize(*buff_size*, *int_size*)
Return the index buffer size for reading 2-electron integrals

ijkl(*i*, *j*, *k*, *l*)
Based on the four orbital indices i,j,k,l return the address in the 1d vector.

read_ci_coeffs()
Read CI coefficients from file NFT12 and return them as a numpy array of floats.

readseq(*buff_size=15000*, *int_size=8*, *mos=False*, *skip_first=False*)
Read FORTRAN sequential unformatted file with two-electron quantities:

- two electron integrals over AO's: .F08 file
- two electron integrals over MO's: .F09 file
- elements of the two particle density matrix: .F15 file

Args:

buff_size [int] size of the buffer holding values to be read, in gamessus it is stored under `NINTMX` variable and in Linux version is equal to 15000 which is the default value

large_labels [bool] a flag indicating if large labels should were used, if largest label `i` (of the MO) is `i<255` then `large_labels` should be `False` (case `LABSIZ=1` in gamessus), otherwise set to `True` (case `LABSIZ=2` in gamess(us),

skip_first [bool] skips the first record of the file is set to True,

Returns: numpy 1D array holding the values

factor(*i*, *j*, *k*, *l*)
Based on the orbitals indices return the factor that takes into account the index permutational symmetry.

ijkl(*i*, *j*, *k*, *l*)
Based on the four orbital indices i, j, k, l return the address in the 1d vector.

print_twoe(*twoe*, *nbf*)
Print the two-electron integrals.

rec
alias of `chemtools.calculators.gamessreader.record`

tri2full(*vector*, *sym=True*)
Convert a triagonal matrix whose elements are stored in the *vector* into a rectangular matrix of the shape given by *shape* tuple.

CHAPTER 2

Contributing

- Source
- Report a bug
- Request a feature
- Submit a pull request

CHAPTER 3

Contact

Łukasz Mentel

- github: lmmentel
- email: lmmentel <at> gmail.com

CHAPTER 4

Citing

If you use *chemtools* in a scientific publication, please consider citing the software as

> Łukasz Mentel, *chemtools* – A Python toolbox for computational chemistry, 2014– . Available at: https://github.com/lmmentel/chemtools.

Here's the reference in the BibLaTeX format

```
 @software{chemtools2014,
    author = {Mentel, Łukasz},
    title = {{chemtools} -- A Python toolbox for computational chemistry},
    url = {https://github.com/lmmentel/chemtools},
    version = {0.9.2},
    date = {2014--},
}
```

or the older BibTeX format

```
@misc{chemtools2014,
   auhor = {Mentel, Łukasz},
   title = {{chemtools} -- A Python toolbox for computational chemistry, ver. 0.9.2},
   howpublished = {\url{https://github.com/lmmentel/chemtools}},
   year  = {2014--},
}
```

CHAPTER 5

Funding

This project was realized through the support from the National Science Center (Poland) grant number UMO-2012/07/B/ST4/01347.

CHAPTER 6

Related projects

cclib A python library for parsing

pygamess Python wrapper for Gamess(US)

CHAPTER 7

License

The MIT License (MIT)

Copyright (c) 2014 Lukasz Mentel

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Bibliography

[Schelegel1995] Schlegel, H. B., & Frisch, M. J. (1995). “Transformation between Cartesian and pure spherical harmonic Gaussians”. International Journal of Quantum Chemistry, 54(2), 83–87. doi:10.1002/qua.560540202

[Varandas2007] Varandas, A. J. C. (2007). “Extrapolating to the one-electron basis-set limit in electronic structure calculations. The Journal of Chemical Physics, 126(24), 244105. doi:10.1063/1.2741259

Python Module Index

C

Index

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z