dish

Release 0.1.0

Christopher Mertens

Jun 19, 2024

CONTENTS:

dish - A Lightweight **DI**rac **S**olver for **H**ydrogen-like Systems

CHAPTER

ONE

WHAT IS DISH?

dish is a Python package which allows the simple calculation of wavefunctions and energy levels for Hydrogen-like systems. The calculation is done by default in a relativistic context by solving the Dirac equation for a single electron in a spherical symmetric potential.

A subpackage also allows to retrieve the non-relativistic wavefunctions and energy-levels by solving the Schrödinger equation.

CHAPTER

TWO

GETTING STARTED

To set your system up to work with *dish* see *Setup*.

For a short introduction covering the most basic features see *Quickstart*.

For further information see the *Introduction into dish* or read the *API* documentation.

2.1 Setup

dish requires CPython 3.7 or above but below 3.12 (since gympy2 is not working with Python3.12). At the moment it is required to build the package and install it afterwards from the local sources.

2.1.1 Building the package

On Unix-based systems

1. Make sure to have the following tools installed:
   - A working Python (version < 3.12) solution and the python virtual environment package venv (on Unix systems this needs to be installed separately).
   - The following Python packages pip, build
   - *(optionally)* For increased performance it is highly recommended to compile an included fortran script. For that the following is required
     1. The Fortran compiler gfortran
     2. The Python Package numpy

Note: On Ubuntu the installation can be done via:

```
sudo apt install python3 python3-pip python3-venv python3-numpy gfortran
python3 -m pip install build
```

2. Clone or download this repository.
3. In a shell in the repositories directory run `make`. The Python package will be built in the *dist*-directory.

Note: The build script will detect your installation of your python installation using `which python3` and therefore also works in virtual python environments when they are activated in your current shell. If you wish to use a specific python executable modify line 3 in the Makefile:

```
# old version
#PY := $(shell which python3)
# modified version
PY := /path/to/your/executable
```

If you encounter any problems using *make* you can build it by hand by running the following commands in the main directory:

```
mkdir dist
cp -r src/ dist/
cp LICENSE MANIFEST.in pyproject.toml README.md dist/
cd dist/src/dish/util/numeric
python3 -m numpy.f2py -c adams_f.f90 -m adams_f -llapack
# if this fails to identify your fortran compiler use
#  --f90exec=/path/to/gfortran --f77exec=/path/to/gfortran
# as additional arguments

cd ../../../../..

python3 -m build dist --outdir dist
```

Note: This assumes you have a working Python environment (with Python < 3.12 (!)) with numpy installed.

On a Windows system

1. Make sure to have the following tools installed:
    1. A working Python solution.
    2. The following Python packages pip, build, meson
2. Clone or download this repository.
3. Change into the repositories directory and run

    ```
    python -m build --outdir dist
    ```

    The Python package will be built in the *dist*-directory.

Note: The package is named *qm-dish* for installation to be distinguished from another package called dish which appears to be a shell implementation for Unix-systems.

2.1.2 Installing the package

On **Unix** based systems after building the package run in the same directory

```
make install
```

Note: This will install the package using pip. An automatic installation using conda is currently not supported but building the package also provides the *sdist* from which a conda version can be build.

On a **Windows** system (*or if you encounter any problems*) run from the same directory

```
python -m pip install qm-dish --find-links dist/
```

The freshly built package and the required dependencies will be installed.

2.2 Quickstart

All calculations internally are performed dimensionless using Hartree atomic units. All classes and functions therefore expect values in atomic units. To convert between units of the SI system and atomic units *dish* provides the method `convert_units`.

It returns the converted value of *value* (given in *old_unit*) in *new_unit*. (Simplified this returns `value * new_unit / old_unit`). ! You need to assure that *old_unit* and *new_unit* have the same dimension as there are no checks performed! *old_unit* and *new_unit* can be either numerical values or strings for the most common units (e.g. “E_h”, “eV”, “J”).

To calculate electronic states of a Hydrogen-like system first the properties of the `Nucleus` need to be specified. *dish* has build-in support for either a point-like nucleus (a pure Coulomb potential), a homogeneously charged ball-like nucleus or a nucleus which charge is described by a Fermi distribution:

$$\rho(r) = \frac{\rho_0}{1 + \exp((r - c)/a)}$$

Therefore, to following properties need to be specified:

1. The nuclear charge *Ze* passed to the parameter `Z`.
2. The mass of the nucleus `M`. To perform calculations with a fixed core this can be set to *numpy.inf* .
3. Parameters of the charge distribution. The radius can be either given as the root mean squared radius $R_{\text{rms}}\sqrt{\langle r^2 \rangle}$ via the parameter `R_rms`, for a ball-like model as the radius R_0 via the parameter `R0` or for a Fermi charge density distribution as the parameter c via the constructor parameter of the same name.

   If a value is passed to one of the three parameters the others are calculated if possible. For a point-like model this must not be passed.

   For a Fermi model also the *diffuseness* parameter a is required. It can be passed explicitly to the constructor via the parameter `a` but defaults to $2.3\text{fm}/a_0/(4 \cdot \ln(3))$ as this is a good approximation for most stable nuclei (*Parpia and Mohanty, Phys.Rev.A, 46 (1992), Number 7*).

   Note: For small nuclei the Fermi-parameter c can not be derived from R_{rms} as the model can’t be applied with the default value of a.

An example for Ca19+ looks like:

```
from dish import Nucleus, convert_units

nuc = Nucleus(Z=20,
              R_rms=convert_units("fm", "a_0", 3.4776),
              M=convert_units("u", "m_e", 40.078)
              )
```

Then the electronic state which should be calculated should be specified. This can be either done be passing the quantum numbers a tuple `(n, l, j)` (e.g. *(2, 0, 1/2)*) or by passing the atomic term symbol as a string `"<n><l-alias><j>"` (e.g. *"2s1/2"*). For the latter one there is another possible notation: `<n>[<l>]<+/->` (e.g. *"2s1/2 == 2[0]+*) where the name for the angular-momentum number can be substituted by the value in brackets and as only single electron systems are calculated *j* will always be $l \pm 1/2$ which can be passed by a + or a -. A mixture of the latter two variants is possible.

From the string a *QuantumNumberSet* object will be constructed using the *parse_atomic_term_symbol* function. You can also pass a *QuantumNumberSet* object directly.

For example the $3p_{1/2}$ state can be specified the following way:

```
state = "3p+"
# which is equivalent to
state = "3[2]1/2
```

After specifying this properties one can run the solving algorithm. In a minimal setup the function call to *solve* looks like:

```
from dish import solve
result = solve(nucleus=nuc, state=state)
```

By default, for the nuclear potential the Fermi model is used. To change this pass the alias of the potential model a string to the `potential_model` parameter:

```
result = solve(nucleus=nuc,
               state=state,
               potential_model="Fermi"  # other options are "ball"=="uniform" or
↪"pointlike"=="coulomb"
               )
```

After the solving process the wave function can be obtained from the *SolvingResult* object which is stored in *result* via

```
Psi = result.wave_function
```

2.2.1 The wave function

has the form

$$\psi_{n\kappa m}(\overline{r}) = \frac{1}{r}\begin{pmatrix} i f_{n\kappa}(r)\Omega_{\kappa,m}(\theta,\varphi) \\ g_{n\kappa}(r)\Omega_{-\kappa,m}(\theta,\varphi) \end{pmatrix}$$

where (r, θ, φ) are spherical coordinates and $\Omega_{\kappa,m}$ is a spherical spinor. It is stored in a *RadialDiracWaveFunction* object which stores the array of points where the wave function is evaluated in the field *r*, the large component f_κ in the field *f* and the small component g_κ in the field *g*:

```
Psi.r  # grid points -> stored in a numpy array
Psi.f  # large component values at the grid points -> stored in a numpy array
Psi.g  # small component values at the grid points -> stored in a numpy array
```

Note: To allow further calculations (mainly the calculation of [matrix elements](#calculating-matrix-elements)) in opposite to literature f and g are stored like above (without the factor $\frac{1}{r}$) to minimize loss of accuracy due to unnecessary rounding.

The energy of the *state* is stored in the field *energy*:

```
result.energy
```

2.2.2 Configuring the solving parameters

The solving algorithm uses a finite-differences approach to solve the coupled system of equations for the large and small component. The energy is searched for which both components can be found continuous. To numerically solve the Dirac equation an Adams-Moulton method is used.

The parameters of the solving algorithm are chosen based on the state by default and can be adjusted by the user:

The Grid

The radial grid on which the wave function is evaluated on is an exponential grid of the form

$$r(t) = r_0 \cdot (\exp(h \cdot t) - 1)$$

where t is a linear grid from 0 to N. This information is stored in a *DistanceGrid* object which can be constructed from the parameters `r0`, `h` and `N` or `r_max` which defines the region where the maximum r value is evaluated. It is recommended to regulate N using the other three parameters and not passing it explicitly. The grid can be passed to the solve function via the parameter `r_grid`:

```
from dish import DistanceGrid
result = solve(nucleus=nuc,
               state=state,
               r_grid=DistanceGrid(h=0.005, r0=1e-6, r_max=2)
               # another possibility is to pass this using a dictionary which will be␣
↪parsed internally:
               # r_grid={"h": 0.005, "r0"=1e-6, "r_max"=2}
               # passing N instead of r_max is also possible but more inconvenient
               # r_grid=DistanceGrid(h=0.005, r0=1e-6, N=20000)
               )
```

Note:

The more points the grid contains and therefore the smaller `r0` and `h` are chosen the more computation intensive is the solving process.
For $N \leq 10^5$ the Python version provides enough speed that the solving finishes in under a second of computation time on decent hardware but for larger `N` it is highly recommended to use the Fortran version which speeds up the main part of the solving process by one to two orders of magnitude.

A better grid for integration purposes

To calculate matrix elements using the wave functions one needs to solve a radial integral. Since the grid on which the wave function is evaluated is given due to the method how the wave function is found, one can either integrate on the exponential grid r or on the linear grid t where the latter one is better suited as the numerical methods to integrate on equidistant points are well understood. On equidistant grids Newton-Cotes-formulas are used, where the lowest order method, the so-called trapezoid rule, is the most efficient but also the most inaccurate.

If possible use a `RombergIntegrationGrid` instead of a `DistanceGrid` since it enables to use Romberg's method for approximating the integral which is an improvement of the trapezoidal rule using Richardson extrapolation. But it requires an equidistant grid with a number of points $N = 2^k + 1$ where k is a positive integer.

One can either just construct a `RombergIntegrationGrid` (which is a subclass of `DistanceGrid`) directly

```
from dish.util.radial.grid import RombergIntegrationGrid

r_grid = RombergIntegrationGrid(h=1e-5,
                                r0=1e-8,
                                k=15,
                                # or N can be passed but needs to fulfill the␣
↪requirement N = 2^k+1
                                # N=2**15+1
                                )
```

or a similar grid can be derived from a given `DistanceGrid` *grid*

```
r_grid = RombergIntegrationGrid.construct_similar_grid_from_distance_grid(grid)
```

where the `h` parameter is altered so that `r_max` and `r0` are kept constant but there is the right number of points for Romberg integration. Note that this method will always create a grid where `h` is smaller (or equal) than in the original grid so that there are more or the same number of points.

The Adams-Moulton method

Adams-Moulton (AM) methods are a linear multistep method to solve systems of ordinary differential equations numerically on a finite grid.
A multistep method of order k uses the information if the previous k points to determine the next point (for a detailed discussion see Chapter (???) of the underlying thesis). Therefore, the AM method requires k initial values.
In the algorithm the AM method is used from the most inner points in the outward direction and inwards from the most outer points.
The initial values are calculated from asymptotic considerations. The order of the method to retrieve the initial values for the inward integration can be specified by the parameter `order_indir`.
The order of the AM itself can be set via `order_AM`.

```
result = solve(nucleus=nuc,
               state=state,
               order_AM=9,    # default value that has proven to be reliable
               order_indir=7  # default value
               )
```

Usually there is no need to modify this values but for some wave functions that oscillate quickly near the origin it might help to increase the order of the AM method.

Note: The higher `order_AM` the more compute intensive is the solving process.

The `indir` function

To use a k^{th} order multistep method k initial values are required in each direction. The initial guesses for the inward direction are obtained from asymptotic assumptions and the wave function is expanded in a converging series (see chapter ?? of the thesis). This series converges very fast if r is large enough in the most outward region and hence only the first few terms need to be calculated. The number of terms can be passed via the parameter `order_indir`. The default value of 7 should be sufficient in most cases.

2.2.3 Functions/Classes used in this Quickstart

`dish.dirac.solver.`**`solve`**(*nucleus: Nucleus*, *state: str | QuantumNumberSet | Tuple[int, int, float]*, *r_grid: DistanceGrid | dict = {'h': 0.005, 'r0': 2e-06}*, *potential_model: str = 'Fermi'*, *m: float = 1*, *E_guess: float | str = 'auto'*, *order_AM: int = 5*, *order_indir: int = 7*, *max_number_of_iterations=20*) → *SolvingResult*

Solve the radial Dirac equation for Hydrogen-like atoms in state 'state'.

Parameters

- **`nucleus`** – parameters of the nucleus
- **`state`** – electron state to find the wave function
- **`r_grid`** – grid on which the wave function is to be evaluated. Can be constructed from dict.
- **`potential_model`** – model of the charge distribution of the nucleus. Either 'Fermi', 'uniform' (charged ball) or 'point-like'. The default is 'Fermi'.
- **`m`** – mass of the particle for which the SE is solved in m_e. The default is 1 (for an electron).
- **`E_guess`** – initial guess for the energy of the state. Can be 'auto' which will use the analytical value for a point-like nucleus. The default is 'auto'.
- **`order_AM`** – order of the Adams-Moulton procedure which is used to solve the differential equation. The default is 5.
- **`order_indir`** – order of the procedure which is used to find the most inner points of the wave function. The default is 7.
- **`max_number_of_iterations`** – number of iterations after which the solving routine will stop. The default is 20.

Returns

The result of the Dirac solving routine 'master' and additional information about the solving process.

Return type

SolvingResult

`dish.util.atomic_units.`**`convert_units`**(*old_unit: str | float*, *new_unit: str | float*, *value=1.0*, *old_unit_exp=1*, *new_unit_exp=1*)

Convert 'value' in unit 'old_unit' to 'new_unit'.

! You need to assure that these values are of the same dimension !

Parameters

- `old_unit` – either the old units name or the conversion value into SI units
- `new_unit` – either the new units name or the conversion value into SI units
- `value` – value to convert. The default is 1 to just obtain the conversion factor.
- `old_unit_exp` – exponent of the old unit. The default is 1.
- `new_unit_exp` – exponent of the new unit. The default is 1.

Returns
converted value

`class dish.util.atom.Nucleus`(*Z: float*, *M: float = inf*, *R_rms: float | None = None*, *R0: float | None = None*, *c: float | None = None*, *a: float = 9.890591475522296e-06*, *system_charge: float | None = None*)

Dataclass to store the properties of the nucleus.

The potential for point-like, ball-like and Fermi charge distributions is provided through the `potential()` method.

Parameters

- `Z` – number of protons in the nucleus == nuclear charge
- `M` – (optional) mass of the nucleus.
- `R_rms` – (optional) root mean squared charge radius of the nucleus. If this is specified
- `R0` – (optional) radius of a homogeneously charged sphere. Usually calculated from R_rms.
- `c` – (optional) Fermi charge distribution parameter *c*. Usually calculated from the R_rms.
- `a` – (optional) diffuseness parameter for a Fermi distribution. Defaults to $2.3\mathrm{fm}/a_0/(4{\cdot}\ln(3))$ as described in *Parpia and Mohanty, Phys.Rev.A, 46 (1992), Number 7*.
- `system_charge` – (optional) charge of the hydrogenic atom. Defaults to *Z-1*.

`potential`(*r: float | ndarray*, *model='Fermi'*)

Parameters

- `r` – point(s) where the value of the potential should be evaluated
- `model` – (optional) alias for the type of the potential model. Choose from "Fermi", "ball-like", "point-like"=="Coulomb". Defaults to "Fermi".

Returns
Values of the nucleus' potential in distance *r*

Return type
float or np.ndarray, same as *r*

`class dish.util.atom.QuantumNumberSet`(*n*, *l=None*, *j=None*, ***, *kappa=None*)

A container class to store the quantum numbers n, l and j. κ is calculated for convenience.

Parameters

- `n` – principal quantum number
- `l` – orbital angular momentum quantum number

- `j` – (optional) total angular momentum quantum number. For non-relativistic calculations not defined.

`dish.util.atom.`**`parse_atomic_term_symbol`**`(`*symbol: str*`)` → *QuantumNumberSet*

Parse the atomic term symbol into quantum numbers.

Parameters

`symbol` – spectroscopic representation of the electronic state

format: '<n><l><j>', where <n> is the integer value, <l> can be either the spectroscopic symbol (s,p,d,…) or an integer in brackets ([1], [2], …) and <j> can be either an explicit integer/2 or +/- .

Example

"2s1/2" == "2[0]+", "15d3/2" == "15[2]-"

Return type

QuantumNumberSet

class `dish.util.misc.`**`SolvingResult`**`(`*state:* dish.util.atom.QuantumNumberSet, *nucleus:* dish.util.atom.Nucleus, *potential_model: str*, *m: float*, *r_grid:* dish.util.radial.grid.grid.DistanceGrid, *wave_function: dish.util.radial.wave_function.RadialWaveFunction*, *energy: float*, *energy_convergence: list*, *solving_parameters: dish.util.misc.SolvingParameters*, *number_of_iterations: int*, *solving_time: float*`)`

class `dish.util.radial.grid.`**`DistanceGrid`**`(`*h: float*, *r0: float*, *N: int | None = None*, *r_max: float | None = None*`)`

Class to hold the information about the grid and the grid points itself. The grid is constructed lazily, i.e. the grid points are calculated the first time they are required. The grid has the form $r(t) = r0 \cdot (\exp t - 1)$ where $t(i) = i \cdot h$ is a linear grid with `N` points. The number of values `N` can be constructed from the maximal r value `r_max`.

class `dish.util.radial.grid.`**`RombergIntegrationGrid`**`(`*h: float*, *r0: float*, *N: int | None = None*, *k: int | None = None*`)`

Class to hold the information about the grid and the grid points itself. The grid is constructed lazily, i.e. the grid points are calculated the first time they are required. The grid has the form $r(t) = r0 \cdot (\exp t - 1)$ where $t(i) = i \cdot h$ is a linear grid with `N` points.

The number of grid points `N` needs to be of the form $N = 2^k + 1$ where k is a positive integer. By using this number of grid points this grid is suited to perform Romberg integration on it which yields more accurate results than the naive trapezoidal rule.

2.3 Introduction into dish

dish is a lightweight solver the Schrödinger and Dirac equation for hydrogenic systems. It allows to find the energy and the wave function for states in these systems and provides a framework to calculate some basic matrix elements.

While *dish* is capable of both relativistic and non-relativistic calculations, the main use case are computations in the relativistic framework and therefore the focus of this documentation will be on these. Working with non-relativistic wave functions will in the most cases work as a drop-in replacement (except for the mismatch in dimensions), but nevertheless the api is discussed in the *last section*.

2.3.1 Hartree Atomic Units

All calculations are performed internally dimensionless in Hartree atomic units. All classes and functions therefore expect values in atomic units. *dish* provides a utility function `dish.util.atomic_units.convert_units()` to convert from and to atomic units. More information *below*.

2.3.2 Defining the Hydrogen-like System

The hydrogenic system is defined by the properties of the nucleus and the model used to describe its potential. There are three potential models implemented:

1. A point-like nucleus, which results in a pure Coulomb potential.

$$\rho(r) = Z\delta(r)$$

2. A ball-like nucleus, which is modeled by a homogeneously charged sphere.

$$\rho(r) = \begin{cases} \rho_0 & , r \leq R_0 \\ 0 & , r > R_0 \end{cases}$$

3. A nucleus with a charge density distribution described by a Fermi distribution:

$$\rho(r) = \frac{\rho_0}{1 + \exp((r - c)/a)}$$

To store the information about the nucleus a `dish.util.atom.Nucleus`-object is used:

```python
from dish.util.atom import Nucleus

nuc = Nucleus(Z:float, M:float=<optional>,
              R_rms:float=<optional>,
              R0:float=<optional>,
              c:float=<optional>, a:float=<optional>,
              system_charge:float=<optional>)
```

The parameters are:

- `Z`: The number of protons which is the nuclear charge.
- `M`: The mass of the nucleus. This is used to take in account for nuclear recoil for non-relativistic calculations. Use *numpy.inf* to have a static nucleus.
- The radius of the charge distribution (not defined for point-like model $\rho(r) = Z\delta(r)$ and therefore optional): It can be passed either via the root-mean-square radius $R_{\text{rms}} = \sqrt{\langle r^2 \rangle}$ via the parameter `R_rms` or via model specific parameters:

  2. For a homogeneously charged sphere the radius R_0 can be passed via the parameter `R0`.
  3. For a Fermi charge distribution the Fermi parameter c be be passed by the correspondingly named parameter `c`.

  Either `R_rms`, `R0` or `c` can be given (but not more than one) and the other parameters are calculated.

  Note: For small values of R_{rms} value the Fermi model is not applicable with the default value of a. If c can't be calculated from R_0 or R_rms an error will be thrown if a Fermi-like model is requested.

- (optional) `a`: To specify the Fermi distribution this *diffuseness* parameter is required. It defaults to $2.3\text{fm}/a_0/(4 \cdot \ln(3))$ as described in *Parpia and Mohanty, Phys.Rev.A, 46 (1992), Number 7*

- (optional) `system_charge`: The charge of the hydrogenic system. This defaults to `Z-1`.

The charge density and the potential can be evaluated at specific points by calling

```
rho = nuc.charge_density(r: float|numpy.ndarray|Distancegrid, model:str=<optional>)

V = nuc.potential(r: float|numpy.ndarray|Distancegrid, model:str=<optional>))
```

To the argument `r` a *float*, *numpy.ndarray* of floats or a `dish.util.grid.DistanceGrid` should be passed to evaluate the charge density or potential at the given points. The argument `model` is optional and expects a *string* that specifies, which model for the nucleus should be used. Valid options are (case-insensitive):

1. For a point-like nuclear model: "point", "point-like", "pointlike", "p", "coulomb", "c"
2. For a homogeneously charged sphere: "u", "uniform", "ball", "uniformball"
3. For a Fermi charge distribution: "f", "fermi"

The potential models and their charge distributions are implemented in *`dish.util.potential`*. To each model there are corresponding classes for the potential and the charge density distribution, which are subclasses of *`dish.util.potential.PotentialModel`* and *`dish.util.potential.ChargeDistribution`* respectively.

Note: It is easily extendable to a custom potential by implementing a subclass of *`dish.util.atom.Nucleus`* and overriding the `potential()` (and optionally the `charge_density()`) method:

```
from dish.util.atom import Nucleus

class MyCustomNucleus(Nucleus):

    def potential(self, r, model):
        if model.lower() == "my-custom-model-name":
            # my implementation
            return result
        else:
            return super().potential(r, model)

    def charge_density(self, r, model):
        if model.lower() == "my-custom-model-name":
            # my implementation
            return result
        else:
            return super().charge_density(r, model)
```

If your nuclear model might be useful for others, you are very welcome to submit a pull-request and implement subclasses of *`dish.util.potential.PotentialModel`* and *`dish.util.potential.ChargeDistribution`* and their respective calls in `Nucleus.potential()`/`Nucleus.charge_density()`.

2.3.3 Defining the Grid

All operations are performed on a finite grid of the form

$$r(t) = r_0 \cdot (\mathrm{e}^t - 1)\,,$$

where t is a linear grid

$$t(i) = i \cdot h\,, \quad i = 0, \ldots, N-1\,.$$

The grid is therefore defined by the parameters r_0, h, N, and is implemented as the class *dish.util.radial.grid.grid.DistanceGrid*.

```
from dish.util.radial.grid.grid import DistanceGrid

grid = DistanceGrid(r0:float, h:float, N:int)
# or
grid = DistanceGrid(r0:float, h:float, r_max:float = <value>)
```

Using the latter method and passing a maximal radial distance N is calculated automatically.

A special grid that is better suited if you want to *calculate matrix elements* of later or calculate any other integral using wave functions from *dish* is a *dish.util.radial.grid.grid.RombergIntegrationGrid*. It has essentially the same structure but the number of grid points is $N = 2^k + 1$ for a positive integer k. This makes integration using Romberg's method possible which yields higher precession most of the times. (See chapter ??? of the underlying thesis for more details.)

It can be constructed by passing `N` or `k` or from a `DistanceGrid` by increasing the number of points by lowering `h` until the condition is met.

```
from dish.util.radial.grid.grid import RombergIntegrationGrid

r_grid = RombergIntegrationGrid(r0:float, h:float, N:int)
r_grid = RombergIntegrationGrid(r0:float, h:float, k:int = <value>)
# or
r_grid = RombergIntegrationGrid.construct_similar_grid_from_distance_
↪grid(grid:DistanceGrid)
```

The actual values of the exponential grid $r[i]$ and the linear grid $t[i]$ can be accessed using

```
r_grid.r  # exponential grid
r_grid.t  # linear grid
r_grid.rp # == r_prime = dr/dt
```

2.3.4 Defining Electronic States

An electronic state in a hydrogenic system is defined by the four quantum numbers n, l, j and m. The orbital angular projection quantum number m is important only for the spherical part of the wave function. The radial part of the wave function is therefore defined by n, l, j. To store these values use an instance of *dish.util.atom.QuantumNumberSet*.

```
from dish.util.atom import QuantumNumberSet

state = QuantumNumberSet(n:int, l:int, j:float)
state = QuantumNumberSet(n:int, kappa:float = <value>)
```

It is common to combine l and j into the Dirac quantum number $\kappa = \mp(j + \frac{1}{2})$, for $j = l \pm \frac{1}{2}$. One can access this as an attribute `state.kappa`.

Usually a state is written not in terms of n, l, j but in spectroscopic notation using a term symbol, e.g. $1s_{\frac{1}{2}}$ instead of $(n, l, j) = (1, 0, \frac{1}{2})$ or $4d_{\frac{3}{2}}$ instead of $(4, 2, \frac{3}{2})$. Since for a hydrogenic system just $j = l \pm \frac{1}{2}$ are possible also a $+$ or $-$ are common, e.g. $1s+ = 1s_{\frac{1}{2}}$ or $4d- = 4d_{\frac{3}{2}}$.

Both versions can be parsed into a `QuantumNumberSet` using *`dish.util.atom.parse_atomic_term_symbol()`*. Alternatively the notation `"n[l]j"` can be used, which is especially useful for higher l, or any combination of these versions.

```
from dish.util.atom import parse_atomic_term_symbol

state: QuantumNumberSet = parse_atomic_term_symbol(state_repr:str)
# e.g.
parse_atomic_term_symbol("1s+")
parse_atomic_term_symbol("4d3/2")
parse_atomic_term_symbol("4[2]-")
```

Note: Even through for s-states only n$s_{\frac{1}{2}}$ exists and n$s_{-\frac{1}{2}}$ does not, it is required to specify the states as "<n>s+" or "<n>s1/2" to distinguish it from *non-relativistic states*.

Having defined the system, the grid and the state one can perform the calculation to find the electronic wave function and the states energy.

2.3.5 Finding the states energy and wave function

```
from dish.schrodinger.solver import solve
from dish.dirac.solver import solve

result = solve(nucleus: dish.util.atom.Nucleus,
               state: dish.util.atom.QuantumNumberSet,
               r_grid: dish.util.radial.grid.grid.DistanceGrid = <optional:dict(r0=1e-6,␣
↪h=1e-4)>,
               potential_model: str = <optional:"Fermi">,
               E_guess: float = <optional:"auto">,
               order_AM: int = <optional:5>,
               order_insch: int = <optional:7>,
               max_number_of_iterations: int = <optional:20>
              )

E = result.energy  # actually E-c^2
wf = result.wave_function  # RadialDiracWaveFunction

wf.f        # large component
wf.g        # small component
wf.r_grid   # grid


from dish.dirac.solver import solve

result = solve(nucleus: dish.util.atom.Nucleus,
```

(continues on next page)

(continued from previous page)

```
               state: dish.util.atom.QuantumNumberSet,
               r_grid: dish.util.radial.grid.grid.DistanceGrid = <optional:dict(r0=1e-6,␣
↪h=1e-4)>,
               potential_model: str = <optional:"Fermi">,
               E_guess: float = <optional:"auto">,
               order_AM: int = <optional:5>,
               order_indir: int = <optional:7>,
               max_number_of_iterations: int = <optional:20>
              )

E = result.energy  # actually E-c^2
wf = result.wave_function  # RadialSchrodingerWaveFunction

wf.R        # radial function
wf.Q        # dR/dr
wf.r_grid   # grid
```

2.3.6 Converting values to atomic units

```
from dish.util.atomic_units import convert_units

convert_units(old_unit: str|float,
              new_unit: str|float,
              value: float = <optional:1.>,
              old_unit_exp: float = 1,
              new_unit_exp: float = 1
             )

# e.g.
convert_units("u", "m_e")
convert_units("J", "E_h")
convert_units("eV", "E_h")
convert_units("m^2", "a_0^2")
convert_units("m", "a_0", old_unit_exp=2, new_unit_exp=2)
```

$$\text{new\_value} = \frac{(\text{value} \cdot \text{new\_unit})^{\text{old\_unit\_exp}}}{\text{new\_unit}^{\text{new\_unit\_exp}}}$$

2.3.7 Calculation of Matrix Elements

```
from dish.util.radial.integration import integrate_on_grid

integrate_on_grid(y:numpy.ndarray, grid:DistanceGrid)
```

$$\langle n_1\kappa_1 \mid O \mid n_2\kappa_2\rangle = \int_0^\infty y(r)dr$$

$$\langle n_1\kappa_1 \mid \gamma_5 \mid n_2\kappa_2\rangle = i\int_0^\infty (-f_1g_2 + g_1f_2)dr$$

```
wf1 = solve(...).wave_function
wf2 = solve(...).wave_function

# assure wf1 and wf2 are evaluated on the same grid
integrate_on_grid(-wf1.f*wf2.g + wf1.g * wf2.f, grid=wf1.grid) * 1j
```

```
from dish.util.radial.operator import BraOperator, SymbolicMatrixOperator

y_5 = SymbolicMatrixOperator([[0,1], [1,0]])

BraOperator(wf1) * y_5 * wf2
```

```
from dish.util.radial.operator import BraOperator, RadialOperator

r_hat = RadialOperator(lambda r: r)

BraOperator(wf1) * r_hat * wf2
```

```
from dish.util.radial.operator import BraOperator, RadialOperator, SymbolicMatrixOperator

r_hat = RadialOperator(lambda r: r)

O = SymbolicMatrixOperator([[1, 1+r_hat], [-5*r_hat, r_hat*r_hat]])

BraOperator(wf1) * O * wf2
```

2.3.8 A High-level Interface for Operators

2.3.9 Non-relativistic Calculations

- electronic state

2.3.10 Functions/Classes used in this introduction

class dish.util.atom.**Nucleus**(*Z: float*, *M: float = inf*, *R_rms: float | None = None*, *R0: float | None = None*, *c: float | None = None*, *a: float = 9.890591475522296e-06*, *system_charge: float | None = None*)

Dataclass to store the properties of the nucleus.

The potential for point-like, ball-like and Fermi charge distributions is provided through the *potential()* method.

Parameters

- **Z** – number of protons in the nucleus == nuclear charge
- **M** – (optional) mass of the nucleus.
- **R_rms** – (optional) root mean squared charge radius of the nucleus. If this is specified
- **R0** – (optional) radius of a homogeneously charged sphere. Usually calculated from R_rms.

- **c** – (optional) Fermi charge distribution parameter c. Usually calculated from the R_rms.
- **a** – (optional) diffuseness parameter for a Fermi distribution. Defaults to $2.3\text{fm}/a_0/(4\cdot\ln(3))$ as described in *Parpia and Mohanty, Phys.Rev.A, 46 (1992), Number 7*.
- **system_charge** – (optional) charge of the hydrogenic atom. Defaults to *Z-1*.

class dish.util.potential.**ChargeDistribution**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**CoulombChargeDistribution**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**CoulombPotential**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**FermiChargeDistribution**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**FermiPotential**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**PotentialModel**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**UniformBallChargeDistribution**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.potential.**UniformBallPotential**(*nucleus: dish.util.atom.Nucleus*)

class dish.util.radial.wave_function.**RadialDiracWaveFunction**(*r_grid: array | DistanceGrid, Psi: array, state: dish.util.atom.QuantumNumberSet*)

Class to store relativistic wave functions for a specific state that solves the radial Dirac equation. Stores the following information: - r_grid: A dish.util.radial.grid.DistanceGrid on which the wave function is evaluated. - Psi: The values of the radial wave function. A two-dimensional array.

- f: The large component.
- g: The small component.

- state: The associated state as a dish.util.atom.QuantumNumberSet.

2.4 Examples

```
import numpy as np
import matplotlib.pyplot as plt

from dish.util.atom import Nucleus, parse_atomic_term_symbol
from dish.util.atomic_units import convert_units
from dish.util.radial.grid.grid import DistanceGrid
from dish.dirac.solver import solve

fig1, ax = plt.subplots(nrows=2, figsize=(6,4), sharex=True)

nuc = Nucleus(Z=1,
            R0=convert_units("m", "a_0", .8783e-15),
            M=np.inf,
            )
grid = DistanceGrid(r0=1e-3, h=1e-3, r_max=150)
res1 = solve(nucleus=nuc, state=parse_atomic_term_symbol("5d-"), r_grid=grid)
ax[0].plot(grid.r, res1.wave_function.f, label="$5d-$")
```

(continues on next page)

(continued from previous page)

```
ax[1].plot(grid.r, res1.wave_function.g)

res2 = solve(nucleus=nuc, state=parse_atomic_term_symbol("3p-"), r_grid=grid)
ax[0].plot(grid.r, res2.wave_function.f, label="$3p-$")
ax[1].plot(grid.r, res2.wave_function.g)

ax[1].set_xlabel("$r$ (in a.u.)")
ax[0].set_ylabel("$f(r)$")
ax[1].set_ylabel("$g(r)$")

ax[0].legend()
```

2.5 API

dish.dirac.master
dish.util.radial.wave_function

2.5.1 dish.dirac.master

Functions

`master`(n, l, j, Z, V, r[, t, h, m_particle, ...])
`outer_classical_turning_point`(V, W)

2.5.2 dish.util.radial.wave_function

Classes

RadialDiracWaveFunction(r_grid, Psi, state)	Class to store relativistic wave functions for a specific state that solves the radial Dirac equation.
`RadialSchrodingerWaveFunction`(r_grid, Psi, state)	Class to store non-relativistic wave functions for a specific state that solves the radial Schrödinger equation.
`RadialWaveFunction`(r_grid, Psi, state)	Base class to store wave functions for a specific state.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX