



# leveldiagram

*Release 0.3.1*

**David Meyer**

**Jan 25, 2024**



# DOCUMENTATION

<b>1</b>	<b>Quick Usage</b>	<b>3</b>
1.1	Introduction . . . . .	5
1.2	Detailed API Documentation . . . . .	8
1.3	Changelog . . . . .	14
1.4	Artist Tests . . . . .	15
1.5	LD Tests . . . . .	22
	<b>Python Module Index</b>	<b>31</b>
	<b>Index</b>	<b>33</b>





This module creates energy level diagrams common to atomic physics as matplotlib graphics. The level structure is defined using networkx graphs.

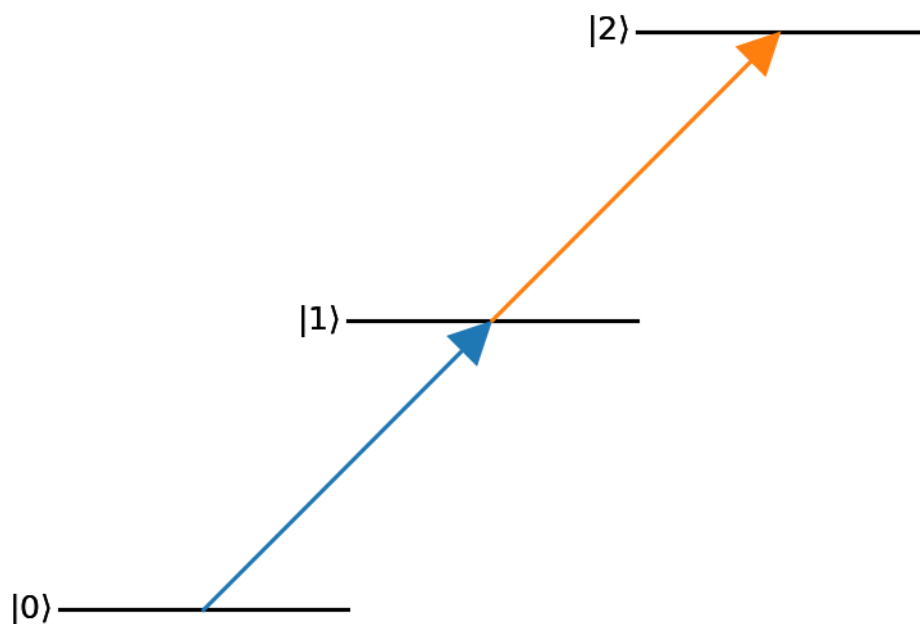


## QUICK USAGE

This package takes networkx directional graphs, which can be used to effectively define a system hamiltonian, and creates an energy diagram representing the system. The nodes of the graph represent the energy levels. The edges of the graph represent the couplings between levels.

Passing a simple graph to the basic level diagram constructor will produce a passable output for simple visualization purposes.

```
nodes = (0,1,2)
edges = ((0,1),(1,2))
graph = nx.DiGraph()
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)
d = ld.LD(graph)
d.draw()
```



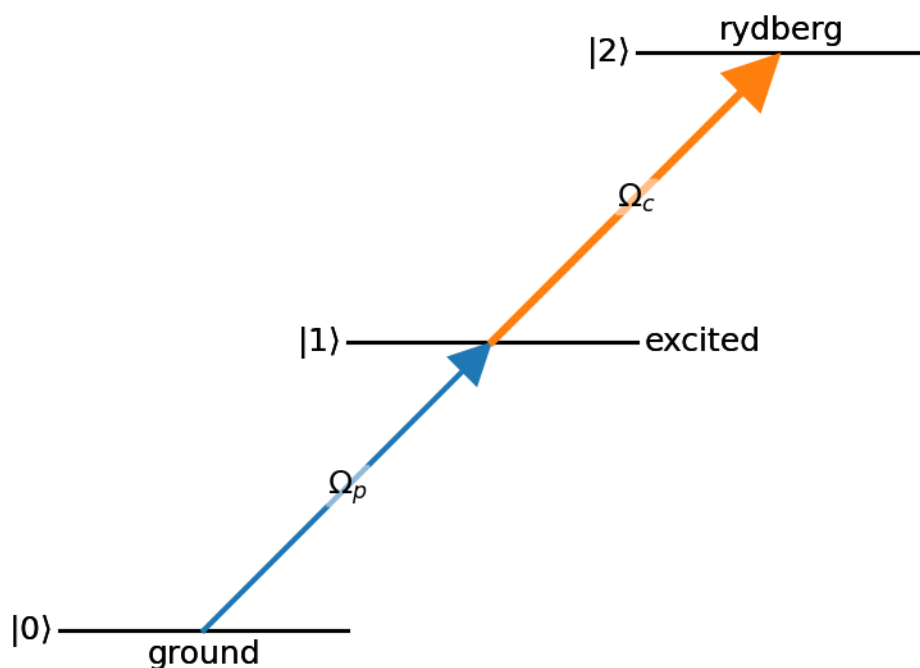
Global settings for the three primitive objects used by levelediagram can be set by passing keyword argument dictionaries to the LD constructor. To control options for a single level or coupling, save these keyword arguments to the respective node or edge of the supplied graph. Generally, the levels and couplings take standard matplotlib 2D line configuration arguments.

```
nodes = ((0,{ 'bottom_text': 'ground' }),
          (1,{ 'right_text': 'excited' }),
          (2,{ 'top_text': 'rydberg' }))
edges = ((0,1, { 'label': '$\\Omega_p$' }),
```

(continues on next page)

(continued from previous page)

```
(1,2, {'label':'$\Omega_c$'})
graph = nx.DiGraph()
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)
d = ld.LD(graph, coupling_defaults = {'arrowsize':0.15,'lw':3})
d.draw()
```



With some basic scripting to create the graph appropriately, much more complicated level diagrams can be made with relative ease.

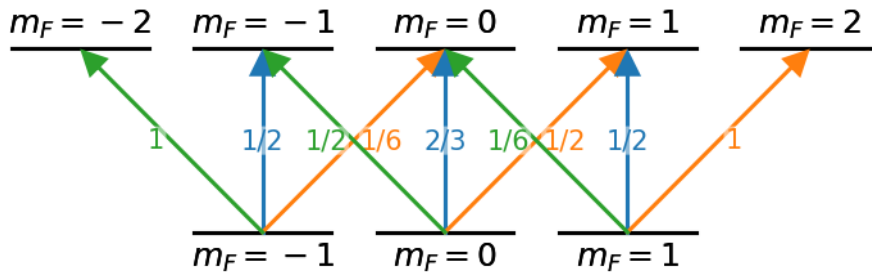
```
hf_nodes = [((f,i), {'top' if f==2 else 'bottom' + '_text':'$m_F='+f'{i:d}'+ '$',
                    'energy':f-1,
                    'xpos':i,
                    'width':0.75,
                    'text_kw':{'fontsize':'large'}})]
for f in [1,2]
for i in range(-f,f+1)]
lin_couples = [((1,i),(2,i),{'label':1,'color':'C0',
                             'label_kw':{'fontsize':'medium','color':'C0'}})]
for i,l in zip(range(-1,2), ['1/2','2/3','1/2'])]
sp_couples = [((1,i),(2,i+1),{'label':1,'color':'C1',
                              'label_offset':'right',
                              'label_kw':{'fontsize':'medium','color':'C1'}})]
for i,l in zip(range(-1,2), ['1/6','1/2','1'])]
sm_couples = [((1,i),(2,i-1),{'label':1, 'color':'C2',
                              'label_offset':'left',
                              'label_kw':{'fontsize':'medium','color':'C2'}})]
for i,l in zip(range(-1,2), ['1','1/2','1/6'])]
hf_edges = lin_couples + sp_couples + sm_couples
hf_graph = nx.DiGraph()
hf_graph.add_nodes_from(hf_nodes)
hf_graph.add_edges_from(hf_edges)
d = ld.LD(hf_graph, default_label = 'none')
d.ax.margins(y=0.2)
```

(continues on next page)



(continued from previous page)

```
d.draw()
```



## 1.1 Introduction

```
%matplotlib inline
```

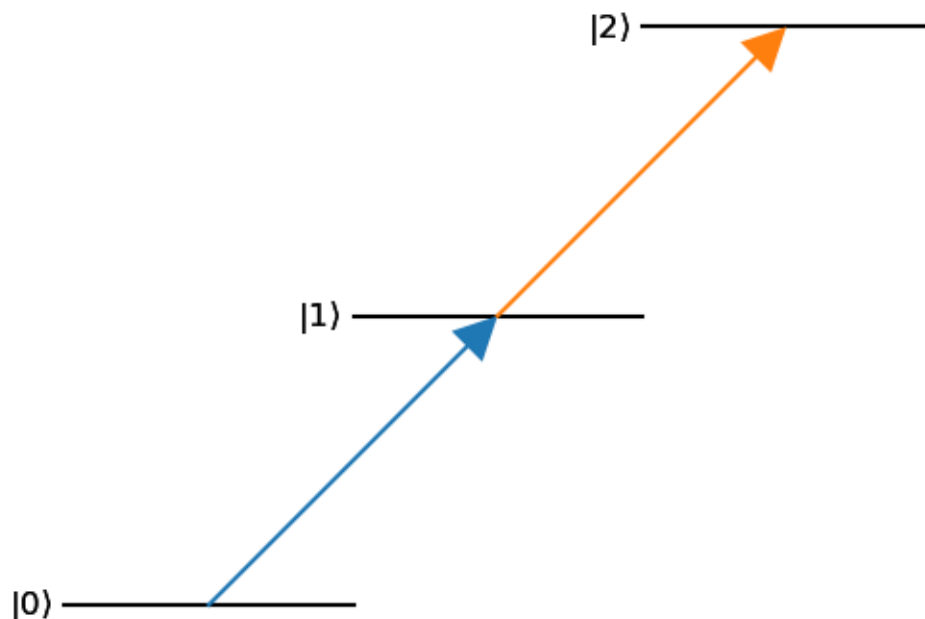
```
import networkx as nx
import leveldiagram as ld
```

To begin, the system is defined using a directional graph, provided by the `networkx.DiGraph` class. The nodes of this graph represent the levels, the edges represent the desired couplings.

Passing a simple graph to the base level diagram constructor `LD` will produce a passable output for simple visualization.

```
nodes = (0,1,2)
edges = ((0,1),(1,2))
graph = nx.DiGraph()
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)
```

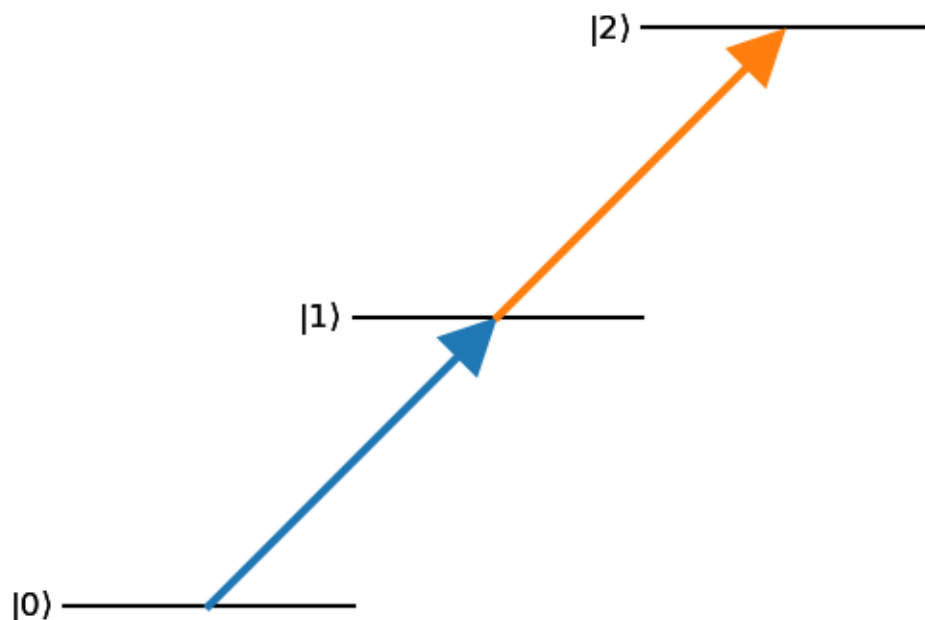
```
d = ld.LD(graph)
d.draw()
d.fig.savefig('basic_example.png', bbox_inches='tight', dpi=150)
```



In keeping with peak matplotlib form, getting something that looks nicer requires applying custom configuration settings that control many of the aspects of the diagram.

Global settings can be controlled by passing in keyword argument dictionaries to the constructor.

```
d = ld.LD(graph, coupling_defaults = {'arrowsize':0.2,'lw':3})
d.draw()
```



NetworkX graphs take an internal structure of nested dictionaries. Leveldiagram utilizes this to provide keyword argument control over each element in the graph.

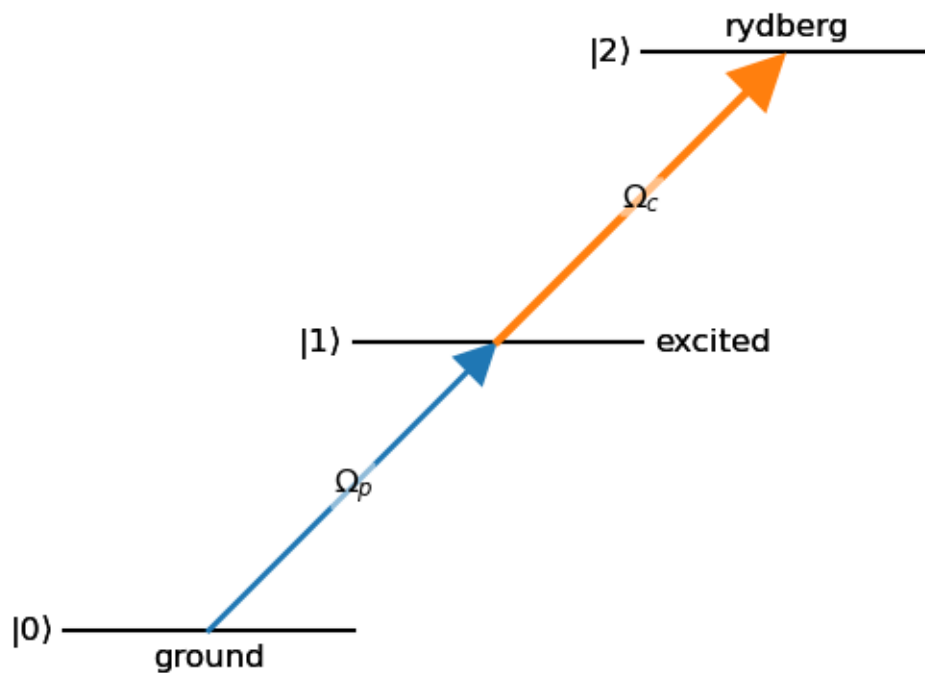
```
nodes = ((0,{'bottom_text':'ground'}),
          (1,{'right_text':'excited'}),
          (2,{'top_text':'rydberg'}))
edges = ((0,1, {'label':'$\Omega_p$', 'lw':2}),
```

(continues on next page)

(continued from previous page)

```
(1,2, {'label':'$\Omega_c$', 'lw':3, 'arrowsize':0.2}))
graph = nx.DiGraph()
graph.add_nodes_from(nodes)
graph.add_edges_from(edges)
```

```
d = ld.LD(graph)
d.draw()
d.fig.savefig('intermediate_example.png', bbox_inches='tight', dpi=150)
```



More detailed examples of element configuration and diagram creation, including embedding a diagram in a larger figure, are shown in the Example Notebooks.

```
ld.about()
```

```
leveldiagram
=====

leveldiagram Version: 0.2.0

Dependencies
=====

Python Version:      3.10.8
NumPy Version:       1.23.4
Matplotlib Version:  3.5.3
NetworkX Version:    2.8.4
```

## 1.2 Detailed API Documentation

Documentation of the classes and methods provided by **leveldiagram**

### 1.2.1 Level Diagram Constructors

```
class leveldiagram.LD(graph, ax=None, default_label='left_text', level_defaults=None,  
                     coupling_defaults=None, wavy_defaults=None, deflection_defaults=None,  
                     use_ld_kw=False)
```

Basic Level Diagram drawing class.

This class is used to draw a level diagram based on a provided Directional Graph. The nodes of this graph define the energy levels, the edges define the couplings.

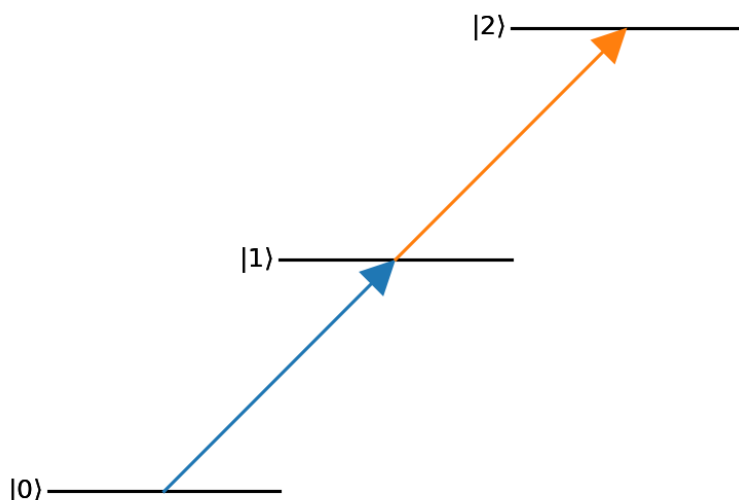
---

**Note:** In keeping with the finest matplotlib traditions, default options and behavior will produce a *reasonable* output from a graph. To get more refined diagrams, global options can be set by passing keyword argument dictionaries to the constructor. Options per level or coupling can be set by setting keyword arguments in the dictionaries of the nodes and edges of the graph.

---

#### Examples

```
>>> nodes = (0,1,2)
>>> edges = ((0,1), (1,2))
>>> graph = nx.DiGraph()
>>> graph.add_nodes_from(nodes)
>>> graph.add_edges_from(edges)
>>> d = ld.LD(graph)
>>> d.draw()
```



#### Parameters

- **graph** (`networkx.DiGraph`) – Graph object that defines the system to diagram

Beyond the arguments provided to the Coupling artist primitive, each coupling plotted by `LD` can also take the following parameters (which are defined as edge attributes on the graph).

- **hidden**: bool - Tells `LD` to ignore this coupling

- **start\_anchor**: str or 2-element tuple - Controls the start anchor point
- **stop\_anchor**: str or 2-element tuple - Controls the stop anchor point
- **detuning**: float - How much to detune the coupling from the transition by. Defined in x-coordinate units.
- **wavy**: bool - Make coupling arrow a sine wave. Uses default options if wavy specific options not provided.
- **deflect**: bool - Make coupling a deflected, circular coupling. Uses default options if deflect specific options not provided.
- **ax** (*matplotlib.axes.Axes*, optional) – Axes to add the diagram to. If None, creates a new figure and axes. Default is None.
- **default\_label** (*str*, optional) – Sets which text label direction to use for default labelling, which is the node index inside a key. Valid options are 'left\_text', 'right\_text', 'top\_text', 'bottom\_text'. If 'none', no default labels are not generated.
- **level\_defaults** (*dict*, optional) – *EnergyLevel* default values for whole diagram. Provided values override class defaults. If None, use class defaults.
- **coupling\_defaults** (*dict*, optional) – *Coupling* default values for whole diagram. Provided values override class defaults. If None, use class defaults.
- **wavy\_defaults** (*dict*, optional) – Wavy specific *Coupling* default values for whole diagram. Provided values override class defaults. If None, use class defaults.
- **deflection\_defaults** (*dict*, optional) – Deflection specific *Coupling* default values for whole diagram. Provided values override class defaults. If None, use class defaults.
- **use\_ld\_kw** (*bool*) –

```
_coupling_defaults = {'arrowsize': 0.15, 'label_kw': {'fontsize': 'large'}}
```

Coupling default parameters dictionary

```
_deflection_defaults = {'deflection': 0.25}
```

Default parameters for a deflection

```
_level_defaults = {'color': 'k', 'text_kw': {'fontsize': 'large'}, 'width': 1}
```

EnergyLevel default parameters dictionary

```
_wavy_defaults = {'halfperiod': 0.1, 'waveamp': 0.05}
```

Default parameters for a wavy coupling

```
couplings: Dict[Tuple[int, int], Coupling]
```

Stores couplings to be drawn

```
draw()
```

Add artists to the figure.

This calls `matplotlib.axes.Axes.autoscale_view()` to ensure plot ranges are increased to account for objects.

It may be necessary to increase plot margins to handle labels near edges of the plot.

```
generate_couplings()
```

Creates the Coupling and WavyCoupling artists from the graph edges.

They are saved to the *couplings* dictionary.

### **generate\_levels()**

Creates the EnergyLevel artists from the graph nodes.

They are saved to the *levels* dictionary.

**levels:** `Dict[int, EnergyLevel]`

Stores levels to be drawn

## 1.2.2 Artist Primitives

Customized matplotlib artist primitives

**class** levelediagram.artists.Coupling(*start, stop, arrowsize, deflection=0.0, waveamp=0.0, halfperiod=0, arrowratio=1, tail=False, arrow\_kw=None, label='', label\_offset='center', label\_rot=False, label\_flip=False, label\_kw=None, \*\*kwargs*)

Bases: `Line2D`

Coupling artist for showing couplings between levels.

This artist is a conglomeration of artists.

- `Line2D` for the actual coupling path
- `Polygon` for the arrow heads
- `Text` for the label

Sufficient methods are overridden from the base `Line2D` class to ensure the other artists are rendered whenever the main artist is rendered.

### **Parameters**

- **start** (*2-element collection*) – Coupling start location in data coordinates
- **stop** (*2-element collection*) – Coupling end location in data coordinates
- **arrowsize** (*float*) – Size of arrowheads in x-data coordinates
- **deflection** (*float, optional*) – Amount to bend the center of the coupling arrow away from linear. Defined in y-coordinates. Default is 0 or no deflection.
- **waveamp** (*float, optional*) – Amplitude of sine wave modulation of the coupling arrow. Defined in y-coordinates. Default is 0 or no waviness.
- **halfperiod** (*float, optional*) – Length of half-period of sine modulation. Defined in x-coordinates. Both **waveamp** and **period** must be non-zero to get modulation. Default is 0 or no waviness.
- **arrowratio** (*float, optional*) – Aspect ratio of the arrowhead. Default is 1 for equal aspect ratio.
- **tail** (*bool, optional*) – Whether to draw an identical arrowhead at the coupling base. Default is False.
- **arrow\_kw** (*dict, optional*) – Dictionary of keyword arguments to pass to `matplotlib.patches.Polygon` constructor. Note that keyword arguments provided to this function will clobber identical keys provided here.
- **label** (*str, optional*) – Label string to apply to the coupling. Default is no label.
- **label\_offset** (*str, optional*) – Offset direction for the label. Options are 'center', 'left', and 'right'. Default is center of the coupling line.
- **label\_rot** (*bool, optional*) – Label is justified along the coupling arrow axis if True. Default is False, so label is oriented along x-axis always.

- **label\_flip** (*bool*, *optional*) – Apply a 180 degree rotation to the label. Default is False.
- **label\_kw** (*dict*, *optional*) – Dictionary of keyword arguments to pass to the `matplotlib.text.Text` constructor.
- **kwargs** – Optional keyword arguments passed to the `matplotlib.lines.Line2D` constructor and the `matplotlib.patches.Polygon` constructor for the arrowhead. Note that 'color' will be automatically changed to 'facecolor' for the arrowhead to avoid extra lines.

#### **draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (`.Artist.get_visible` returns False).

#### **Parameters**

**renderer** (~matplotlib.backend\_bases.RendererBase subclass.) –

#### **Notes**

This method is overridden in the Artist subclasses.

#### **init\_arrowheads**(\*\**kwargs*)

Creates the arrowhead(s) for the coupling as matplotlib polygon objects.

#### **Parameters**

**kwargs** – Optional keyword arguments to pass to the `matplotlib.patches.Polygon` constructor.

#### **init\_label**(*label*, *label\_offset*, *label\_rot*, *label\_flip*, \*\**label\_kw*)

Creates the coupling label text object.

#### **Parameters**

- **label** (*str*) – Label string to apply to the coupling.
- **label\_offset** (*str*) – Offset direction for the label. Options are 'center', 'left', and 'right'.
- **label\_rot** (*bool*) – Label will be justified along the coupling arrow axis if True.
- **label\_flip** (*bool*) – Apply a 180 degree rotation to the label.
- **label\_kw** – Keyword arguments to pass to the `matplotlib.text.Text` constructor.

#### **init\_path**()

Calculates the desired path for the line of the coupling.

The returned path is a line relative to the x-axis of the correct length. Transforms are used to move and rotate this path to the end location. This method of making the couplings is a little convoluted, but it allows for simple definition of very general paths (line sine waves) without distortions.

#### **Returns**

- **x** (*numpy.ndarray*) – x-coordinates of the data points for the un-rotated, un-translated path
- **y** (*numpy.ndarray*) – y-coordinates of the data points for the un-rotated, un-translated path

#### **Return type**

*Tuple[[ndarray](#), [ndarray](#)]*

#### **set\_axes**(*axes*)

**set\_figure**(*figure*)

Set the `.Figure` instance the artist belongs to.

**Parameters**

**fig** (~matplotlib.figure.Figure) –

**set\_transform**(*transform*)

Set the artist transform.

**Parameters**

**t** (~matplotlib.transforms.Transform) –

**class** levelediagram.artists.**EnergyLevel**(*energy, xpos, width, right\_text="", left\_text="", top\_text="", bottom\_text="", text\_kw=None, \*\*kwargs*)

Bases: `Line2D`

Energy level artist.

This object also implements a number of potential Text artists for labelling. It also includes helper methods for getting the exact coordinates of anchor points for connected coupling arrows and the like.

**Parameters**

- **energy** (*float*) – y-axis position of the level
- **xpos** (*float*) – x-axis position of the level
- **width** (*float*) – Width of the level line, in units of the x-axis
- **right\_text** (*str, optional*) – Text to put to the right of the level
- **left\_text** (*str, optional*) – Text to put to the left of the level
- **top\_text** (*str, optional*) – Text to put above the level
- **bottom\_text** (*str, optional*) – Text to put below the level
- **text\_kw** (*dict, optional*) – Dictionary of keyword-arguments passed to `matplotlib.text.Text`
- **kwargs** – Passed to the `matplotlib.lines.Line2D` constructor

**draw**(*renderer*)

Draw the Artist (and its children) using the given renderer.

This has no effect if the artist is not visible (`.Artist.get_visible` returns False).

**Parameters**

**renderer** (~matplotlib.backend\_bases.RendererBase subclass.) –

## Notes

This method is overridden in the Artist subclasses.

**get\_anchor**(*loc='center'*)

Returns an anchor on the level in plot coordinates.

**Parameters**

**loc** (*str or collection of 2 elements*) – What reference point to return. 'center', 'left', 'right' gives those points of the level. A 2-element iterable is interpreted as offsets from the center location.

**Raises**

**TypeError** – If `loc` is not accepted string or a 2-element iterable.

**Return type**

*ndarray*



**get\_center()**

Returns coordinates of the center of the level line.

**Returns**

x,y coordinates

**Return type**

`numpy.ndarray`

**get\_left()**

Returns coordinates of the left of the level line.

**Returns**

x,y coordinates

**Return type**

`numpy.ndarray`

**get\_right()**

Returns coordinates of the right of the level line.

**Returns**

x,y coordinates

**Return type**

`numpy.ndarray`

**set\_axes(*axes*)****Parameters**

**axes** (*Axes*) –

**set\_data(*x*, *y*)**

Set the x and y data.

**Parameters**

**\*args** ((2, *N*) array or two 1D arrays) –

**set\_figure(*figure*)**

Set the .Figure instance the artist belongs to.

**Parameters**

**fig** (~matplotlib.figure.Figure) –

**set\_transform(*transform*)**

Overridden to add padding offsets to labels.

**text\_labels: Dict[str, Text]**

Text label objects to add to the level

## 1.2.3 Utilities

Miscellaneous utility functions

**leveldiagram.utils.about()**

Display version of leveldiagram and critical dependencies.

**leveldiagram.utils.bra\_str(*s*)**

Put a bra around the string in matplotlib.

**Parameters**

**s** (*Any*) – Object to be converted to a string and placed inside a bra.

**Returns**

A string that will render as  $\langle s |$

#### Return type

`str`

`leveldiagram.utils.deep_update(mapping, *updating_mappings)`

Helper function to update nested dictionaries.

Lifted from `pydantic`

#### Returns

Deep-updated copy of mapping

#### Return type

`dict`

#### Parameters

- `mapping` (`dict`) –
- `updating_mappings` (`dict`) –

`leveldiagram.utils.ket_str(s)`

Put a ket around the string in matplotlib.

#### Parameters

`s` (`Any`) – Object to be converted to string and placed inside a ket.

#### Returns

A string that will render as  $|s\rangle$

#### Return type

`str`

## 1.3 Changelog

### 1.3.1 v0.3.1

#### Improvements

- Remove implicit assumption in LD that graph node keys must be integers or floats

### 1.3.2 v0.3.0

#### Improvements

- Many improvements for releasing, docs, and linting workflows

### 1.3.3 v0.2.0

#### Improvements

- All coupling types are now available in `Coupling`
- Added option for circular coupling paths, set by a deflection parameter
- Added many options to LD for working with graphs from other projects
  - Can now specify all leveldiagram control parameters under a single key `ld_kw`. This helps avoid key naming conflicts between projects.
  - Wavy and deflected couplings are enabled via `'wavy'` and `'deflect'` boolean control parameters

- Individual couplings can be ignored by setting 'hidden' to True
- Start and stop anchors can be specified independently for couplings

### Bug Fixes

- Fixed definition when using custom anchor positions

### Deprecations

- WavyCoupling is no longer used. Use Coupling with waveamp and halfperiod parameters defined.

## 1.3.4 v0.1.1

### Improvements

- Add and about function for easy tracking of improvements in example notebooks

### Bug Fixes

- Fixed issue where level labels near the axes edge would get clipped

### Deprecations

- Updated some default plotting values

## 1.3.5 v0.1.0

Initial release.

Includes the artist primitives EnergyLevel, Coupling, and WavyCoupling. Also includes the base leveldiagram creation class LD.

## 1.4 Artist Tests

```
%matplotlib inline
```

```
%load_ext autoreload  
%autoreload 2
```

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

```
from leveldiagram.artists import Coupling, EnergyLevel  
import leveldiagram as ld
```

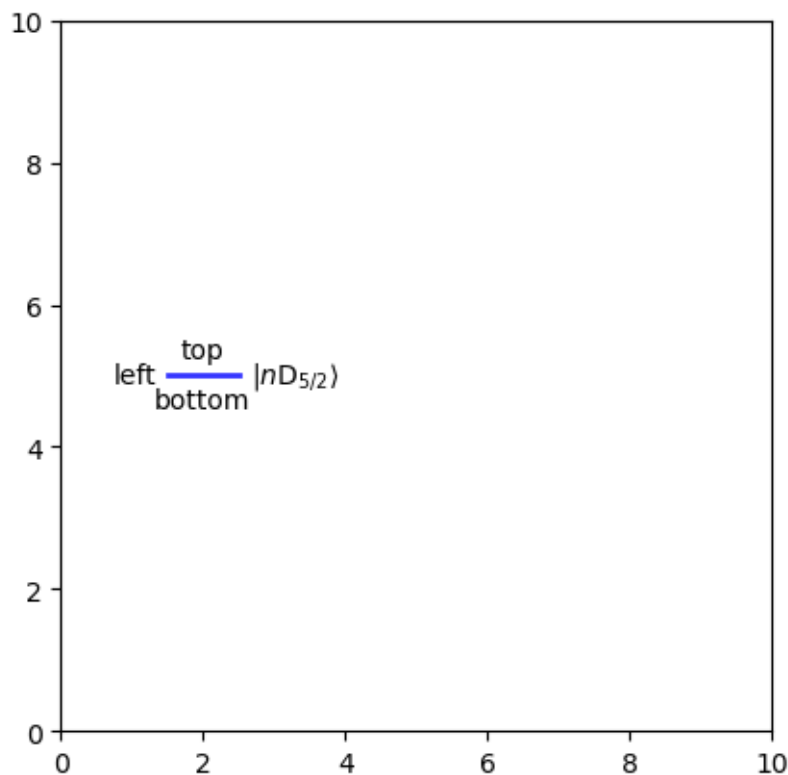
### 1.4.1 Level Tests

```
plt.close('all')
fig, ax = plt.subplots(1)

ax.set_xlim((0,10))
ax.set_ylim((0,10))

x = np.linspace(0,25,151)
y = np.sin(x)
lev = EnergyLevel(5, 2, 1,
                  r'$\left|n\mathrm{D}_{5/2}\right\rangle$', 'left', 'top', 'bottom',
                  color='b',alpha=0.8,linestyle='--', lw=2)
ax.add_line(lev)

ax.set_aspect('equal')
```



### 1.4.2 Coupling Tests

```
plt.close('all')
fig, ax = plt.subplots(1)

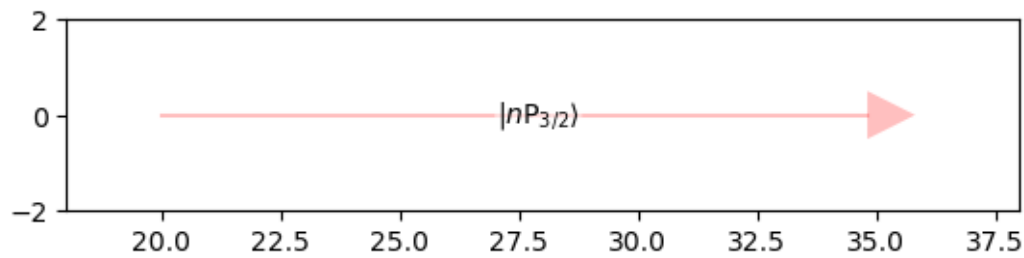
ax.set_xlim((18,38))
ax.set_ylim((-2,2))

ax.add_line(Coupling((20,0),(20+15.81,0),1,color='r',alpha=0.25,linestyle='-',
↪ tail=False, arrow_kw={'ec':'none'},
                    label=r'$|n\mathrm{P}_{3/2}\rangle$', label_offset='center', label_kw=
↪ {'rotation_mode':'default'}))
```

(continues on next page)

(continued from previous page)

```
ax.set_aspect('equal')
```

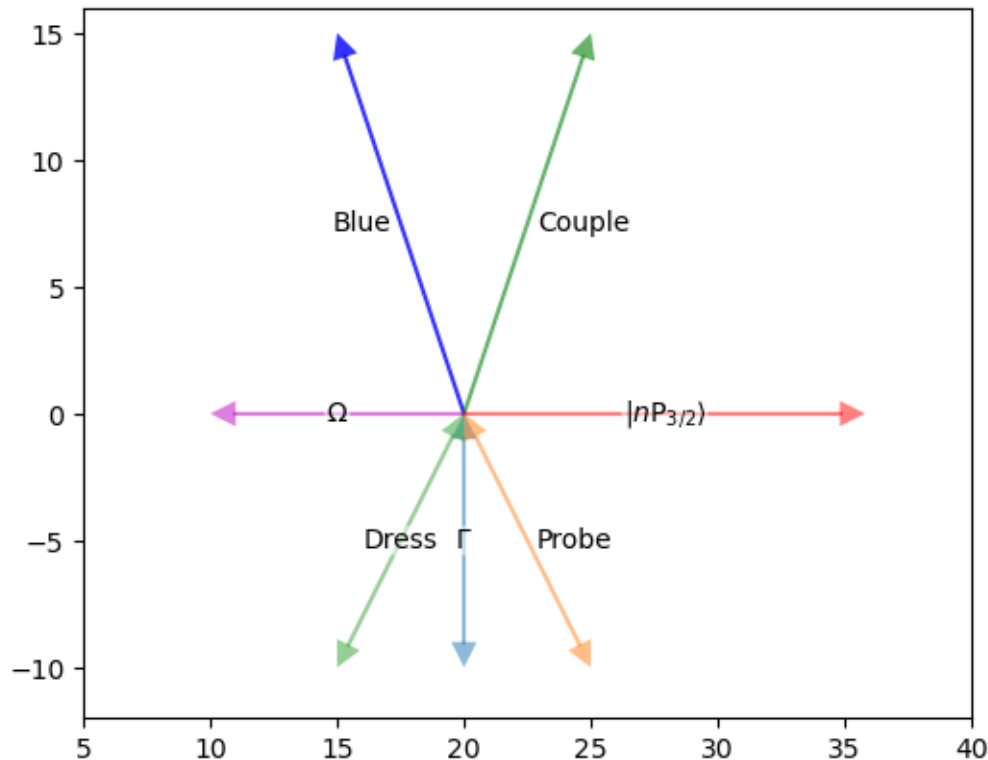


```
plt.close('all')
fig, ax = plt.subplots(1)

ax.set_xlim((5,40))
ax.set_ylim((-12,16))

ax.add_line(Coupling((20,0),(15,15),1,color='b',alpha=0.8,linestyle='-', arrow_kw={'ec
→ ':'none'},
                label='Blue', label_offset='left', label_kw={'rotation_mode':'default
→ '}))
ax.add_line(Coupling((20,0),(25,15),1,color='g',alpha=0.6,linestyle='-',
                label='Couple', label_offset='right', label_kw={'rotation_mode':'default'
→ '}))
ax.add_line(Coupling((20,0),(20+15.81,0),1,color='r',alpha=0.5,linestyle='-', arrow_
→ kw={'ec':'none'},
                label=r'$|n\mathrm{P}_{3/2}\rangle$', label_offset='center', label_kw=
→ {'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(10,0),1,color='m',alpha=0.5,linestyle='-',
                label=r'$\Omega$', label_offset='center', label_kw={'rotation_mode
→ ':'default'}))
ax.add_line(Coupling((20,0),(20,-10),1,tail=True,color='C0',alpha=0.5,linestyle=
→ '-', arrow_kw={'ec':'none'},
                label=r'$\Gamma$', label_offset='center', label_kw={'rotation_mode
→ ':'default'}))
ax.add_line(Coupling((20,0),(25,-10),1,tail=True,color='C1',alpha=0.5,linestyle=
→ '-',
                label='Probe', label_offset='right', label_kw={'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(15,-10),1,tail=True,color='C2',alpha=0.5,linestyle=
→ '-', arrow_kw={'ec':'none'},
                label='Dress', label_offset='center', label_kw={'rotation_mode':'default'
→ '}))

ax.set_aspect('equal')
```

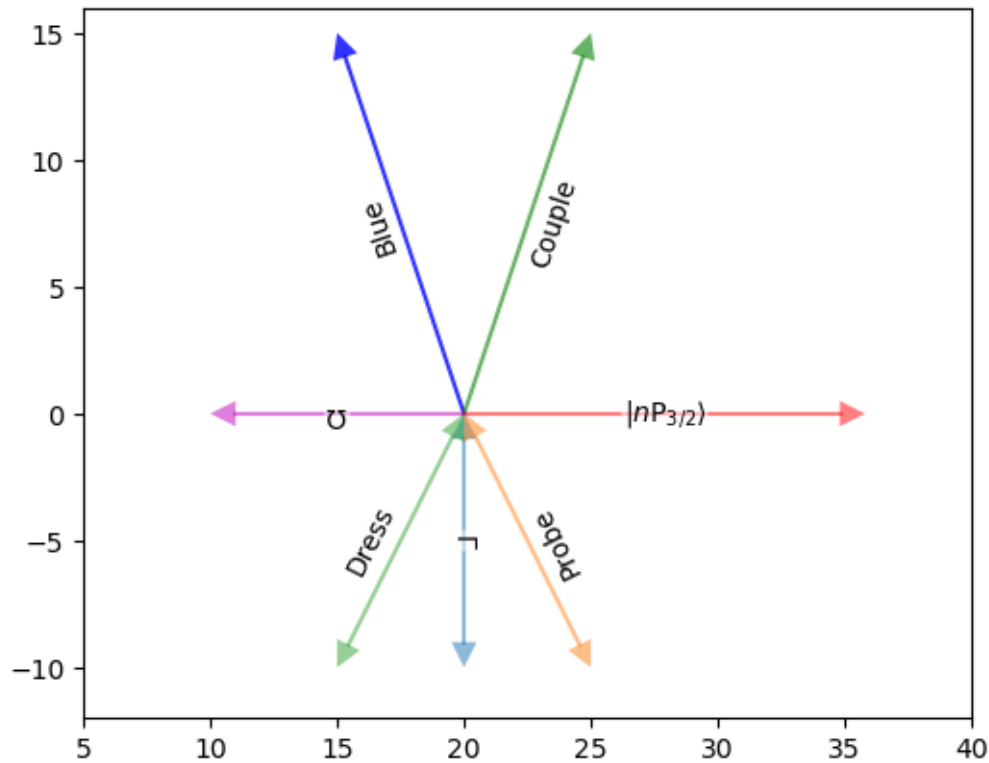


```
plt.close('all')
fig, ax = plt.subplots(1)

ax.set_xlim((5,40))
ax.set_ylim((-12,16))

ax.add_line(Coupling((20,0),(15,15),1,color='b',alpha=0.8,linestyle='-',
                    label='Blue', label_offset='left', label_rot=True, label_kw={
                        'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(25,15),1,color='g',alpha=0.6,linestyle='-',
                    label='Couple', label_offset='right', label_rot=True, label_kw={
                        'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(20+15.81,0),1,color='r',alpha=0.5,linestyle='-',
                    label=r'$|n\mathrm{P}_{3/2}\rangle$', label_offset='center', label_
                        rot=True, label_kw={'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(10,0),1,color='m',alpha=0.5,linestyle='-',
                    label=r'$\Omega$', label_offset='center', label_rot=True, label_
                        kw={'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(20,-10),1,tail=True,color='C0',alpha=0.5,linestyle=
                        '- ',
                    label=r'$\Gamma$', label_offset='center', label_rot=True, label_
                        kw={'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(25,-10),1,tail=True,color='C1',alpha=0.5,linestyle=
                        '- ',
                    label='Probe', label_offset='right', label_rot=True, label_flip=True,
                        label_kw={'rotation_mode':'default'}))
ax.add_line(Coupling((20,0),(15,-10),1,tail=True,color='C2',alpha=0.5,linestyle=
                        '- ',
                    label='Dress', label_offset='left', label_rot=True, label_flip=True,
                        label_kw={'rotation_mode':'default'}))

ax.set_aspect('equal')
```



### 1.4.3 Wavy Coupling Tests

```
plt.close('all')
fig, ax = plt.subplots(1)

ax.set_xlim((0,40))
ax.set_ylim((-20,20))

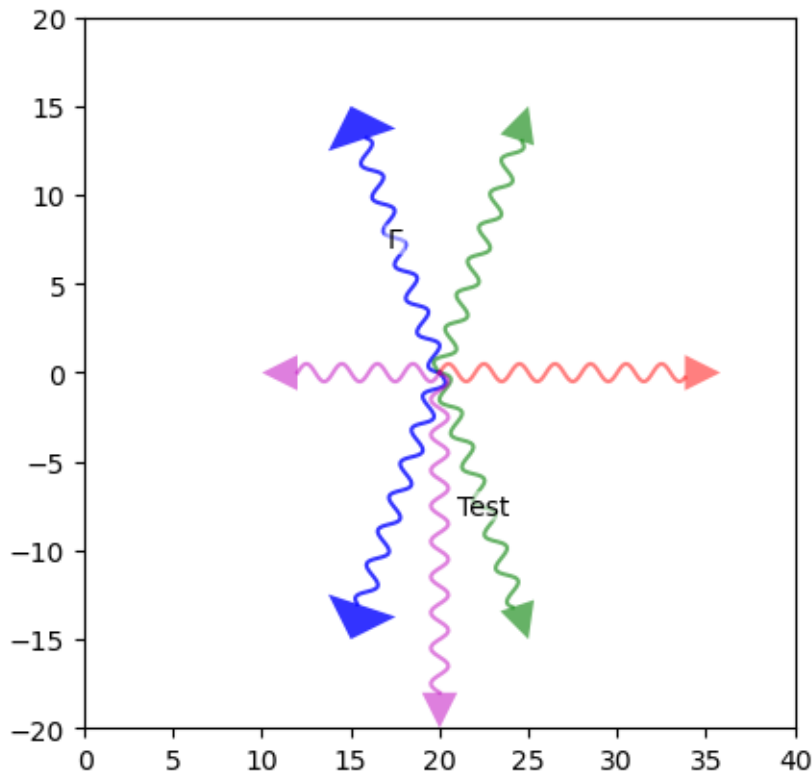
x = np.linspace(0,25,151)
y = np.sin(x)
ax.add_line(Coupling((20,0),(15,15),2,
                    arrowratio=2,
                    waveamp=0.5,halfperiod=1,
                    color='b',alpha=0.8,linestyle='-', label=r'$\Gamma$'))
ax.add_line(Coupling((20,0),(25,15),2,
                    waveamp=0.5,halfperiod=1,
                    color='g',alpha=0.6,linestyle='-'))
ax.add_line(Coupling((20,0),(15,-15),2,
                    arrowratio=2,
                    waveamp=0.5,halfperiod=1,
                    color='b',alpha=0.8,linestyle='-'))
ax.add_line(Coupling((20,0),(25,-15),2,
                    waveamp=0.5,halfperiod=1,
                    color='g',alpha=0.6,linestyle='-', label='Test'))
ax.add_line(Coupling((20,0),(20+15.81,0),2,
                    waveamp=0.5,halfperiod=1,
                    color='r',alpha=0.5,linestyle='-', arrow_kw={'ec':'none'}))
ax.add_line(Coupling((20,0),(10,0),2,
                    waveamp=0.5,halfperiod=1,
                    color='m',alpha=0.5,linestyle='-'))
```

(continues on next page)

(continued from previous page)

```
ax.add_line(Coupling((20,0), (20,-20), 2,
                    waveamp=0.5,halfperiod=1,
                    color='m', alpha=0.5, linestyle='-'))

ax.set_aspect('equal')
```



#### 1.4.4 Elliptic Wavy Coupling Tests

```
plt.close('all')
fig, ax = plt.subplots(1)

ax.set_xlim((0,40))
ax.set_ylim((-20,20))

x = np.linspace(0,25,151)
y = np.sin(x)
ax.add_line(Coupling((20,0),(15,15),2,
                    deflection=2,
                    arrowratio=2,
                    waveamp=0.5,halfperiod=1,
                    color='b',alpha=0.8,linestyle='-', label=r'$\Gamma$'))
ax.add_line(Coupling((20,0),(25,15),2,
                    deflection=3,
                    color='g',alpha=0.6,linestyle='-'))
ax.add_line(Coupling((20,0),(15,-15),2,
                    deflection=2,
                    arrowratio=2,
                    waveamp=0.5,halfperiod=1,
```

(continues on next page)



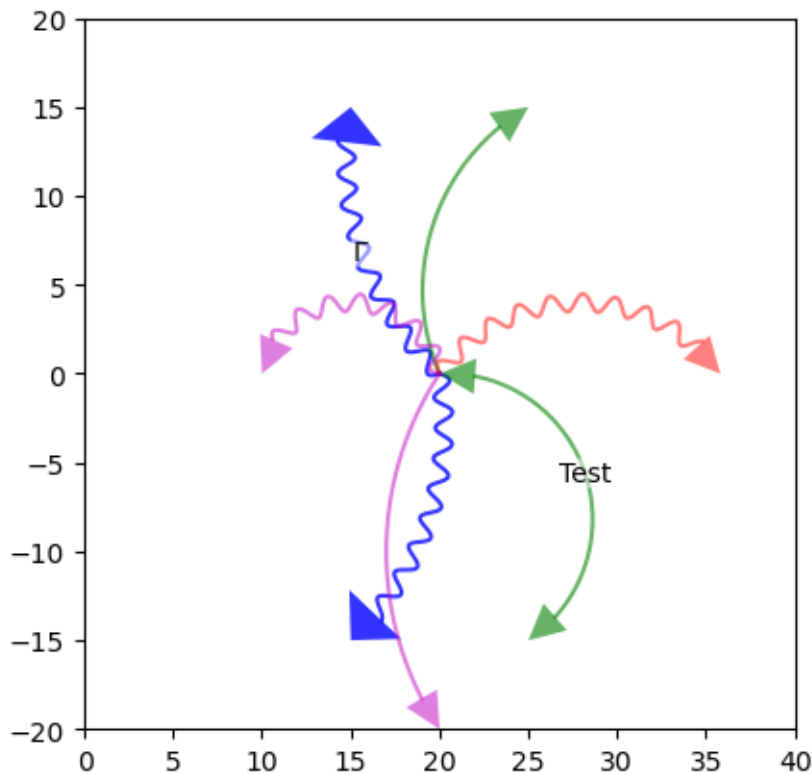
(continued from previous page)

```

        color='b',alpha=0.8,linestyle='-'))
ax.add_line(Coupling((20,0),(25,-15),2,
                    deflection=6,
                    tail=True,
                    color='g',alpha=0.6,linestyle='-', label='Test'))
ax.add_line(Coupling((20,0),(20+15.81,0),2,
                    deflection=4,
                    waveamp=0.5,halfperiod=1,
                    color='r',alpha=0.5,linestyle='-', arrow_kw={'ec':'none'}))
ax.add_line(Coupling((20,0), (10,0), 2,
                    deflection=-4,
                    waveamp=0.5,halfperiod=1,
                    color='m', alpha=0.5, linestyle='-'))
ax.add_line(Coupling((20,0), (20,-20), 2,
                    deflection=-3,
                    color='m', alpha=0.5, linestyle='-'))

ax.set_aspect('equal')

```



```
ld.about()
```

leveldiagram  
=====

```
leveldiagram Version: 0.2.0
```

Dependencies  
=====

(continues on next page)

(continued from previous page)

```
Python Version:      3.10.8
NumPy Version:       1.23.4
Matplotlib Version:  3.5.3
NetworkX Version:    2.8.4
```

## 1.5 LD Tests

```
%matplotlib inline
```

```
%load_ext autoreload
%autoreload 2
```

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

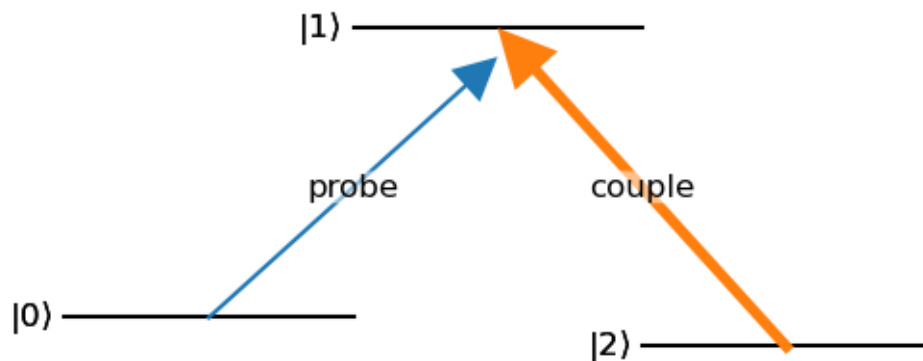
```
import leveldiagram as ld
```

### 1.5.1 Basic 3-level diagrams

#### Lambda

```
lambda_nodes = ((0,
                  (1,
                   (2, {'energy':-0.1})))
lambda_edges = ((0,1,{'detuning':0.1, 'label':'probe'}),
                 (2,1,{'label':'couple', 'lw':4, 'arrowsize':0.2}))
lambda_graph = nx.DiGraph()
lambda_graph.add_nodes_from(lambda_nodes)
lambda_graph.add_edges_from(lambda_edges)
```

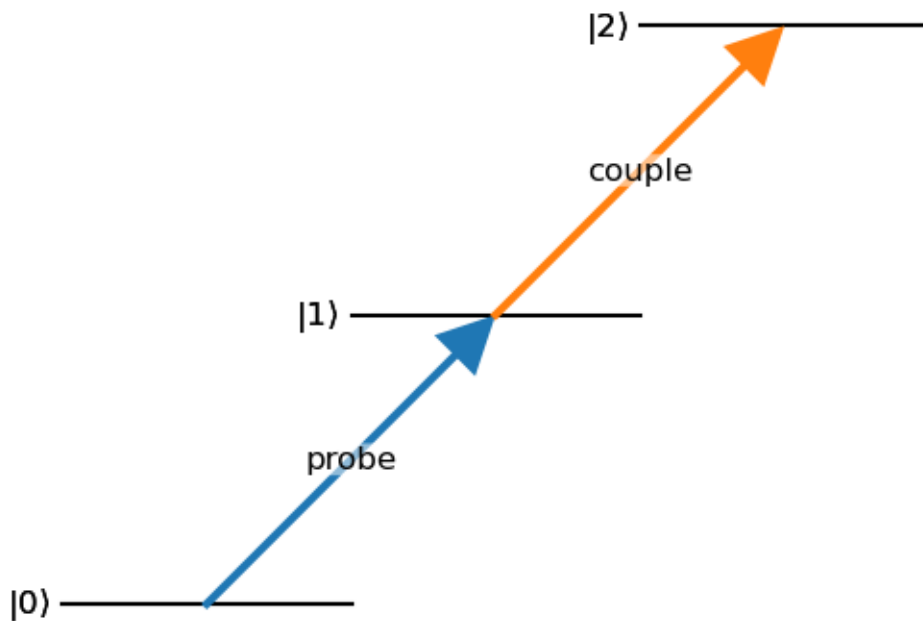
```
d = ld.LD(lambda_graph)
d.draw()
d.fig.savefig('lambda.png', bbox_inches='tight', dpi=150)
```



## Ladder

```
ladder_nodes = (0,1,2)
ladder_edges = ((0,1,{ 'label':'probe' }),
                (1,2,{ 'label':'couple' }))
ladder_graph = nx.DiGraph()
ladder_graph.add_nodes_from(ladder_nodes)
ladder_graph.add_edges_from(ladder_edges)
```

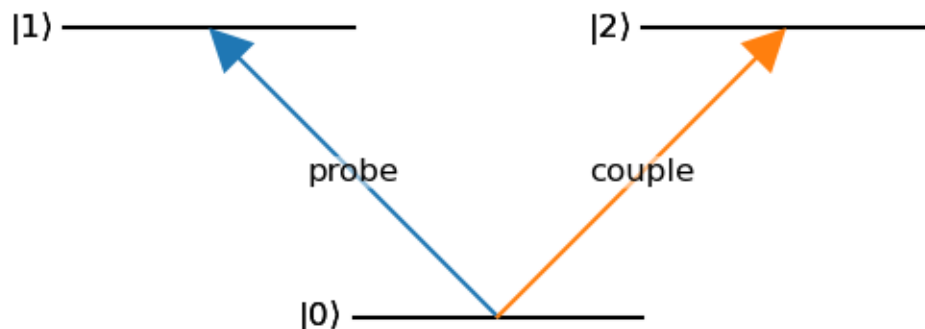
```
d = ld.LD(ladder_graph,
          coupling_defaults = {'arrowsize':0.2,'lw':3})
d.draw()
```



## Vee

```
v_nodes = ((0),
            (1,{ 'energy':1, 'xpos':-1}),
            (2,{ 'energy':1, 'xpos':1}))
v_edges = ((0,1,{ 'label':'probe' }),
            (0,2,{ 'label':'couple' }))
v_graph = nx.DiGraph()
v_graph.add_nodes_from(v_nodes)
v_graph.add_edges_from(v_edges)
```

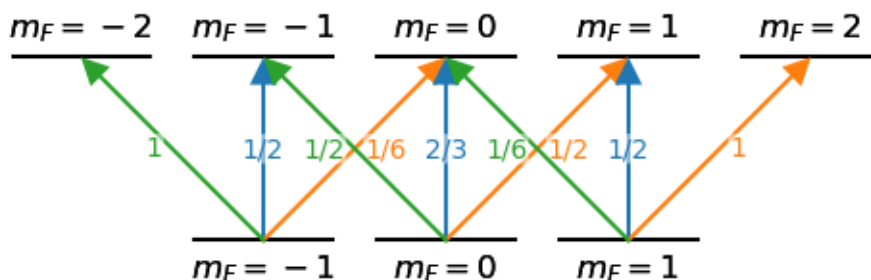
```
d = ld.LD(v_graph)
d.draw()
```



## 1.5.2 Hyperfine Diagram

```
hf_nodes = [((f,i), {'top' if f==2 else 'bottom') + '_text': '$m_F='+f'{i:d}'+ '$',
               'energy':f-1,
               'xpos':i,
               'width':0.75,
               'text_kw':{'fontsize':'large'}})
    for f in [1,2]
    for i in range(-f,f+1)]
lin_couples = [((1,i),(2,i),{'label':l,'color':'C0',
                              'label_kw':{'fontsize':'medium','color':'C0'}})
               for i,l in zip(range(-1,2), ['1/2','2/3','1/2'])]
sp_couples = [((1,i),(2,i+1),{'label':l,'color':'C1',
                              'label_offset':'right',
                              'label_kw':{'fontsize':'medium','color':'C1'}})
              for i,l in zip(range(-1,2), ['1/6','1/2','1'])]
sm_couples = [((1,i),(2,i-1),{'label':l,'color':'C2',
                              'label_offset':'left',
                              'label_kw':{'fontsize':'medium','color':'C2'}})
              for i,l in zip(range(-1,2), ['1','1/2','1/6'])]
hf_edges = lin_couples + sp_couples + sm_couples
hf_graph = nx.DiGraph()
hf_graph.add_nodes_from(hf_nodes)
hf_graph.add_edges_from(hf_edges)
```

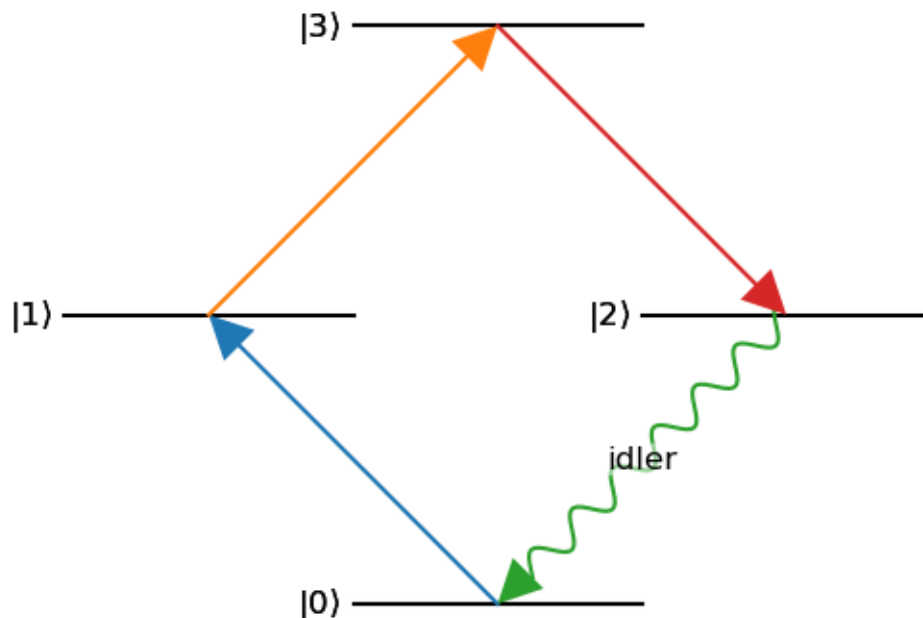
```
d = ld.LD(hf_graph, default_label = 'none')
d.ax.margins(y=0.2)
d.draw()
d.fig.savefig('hyperfine.png', bbox_inches='tight', dpi=150)
```



### 1.5.3 4-wave Mixing Diagram

```
fwm_nodes = ((0),
              (1,{'xpos':-1}),
              (2,{'xpos':1,'energy':1}),
              (3,{'energy':2,'xpos':0}))
fwm_edges = ((0,1),
              (1,3),
              (3,2),
              (2,0,{'label':'idler', 'wavy':True}))
fwm_graph = nx.DiGraph()
fwm_graph.add_nodes_from(fwm_nodes)
fwm_graph.add_edges_from(fwm_edges)
```

```
d = ld.LD(fwm_graph)
d.draw()
```



### 1.5.4 Incorporation into a Larger Figure

```
bx_data = np.linspace(-10, 10, 51)
by_data = 3.2/(bx_data**2 + 2**2)

cx_data = np.linspace(0, 2*np.pi*10, 100)
cy_data = np.sin(cx_data)
```

```
mosaic = [['a', 'b', 'b'],
           ['c', 'c', 'c']]

fig = plt.figure(figsize=(8,4), layout='constrained')
axs = fig.subplot_mosaic(mosaic)

### part a
d = ld.LD(ladder_graph, ax=axs['a'],
```

(continues on next page)

(continued from previous page)

```

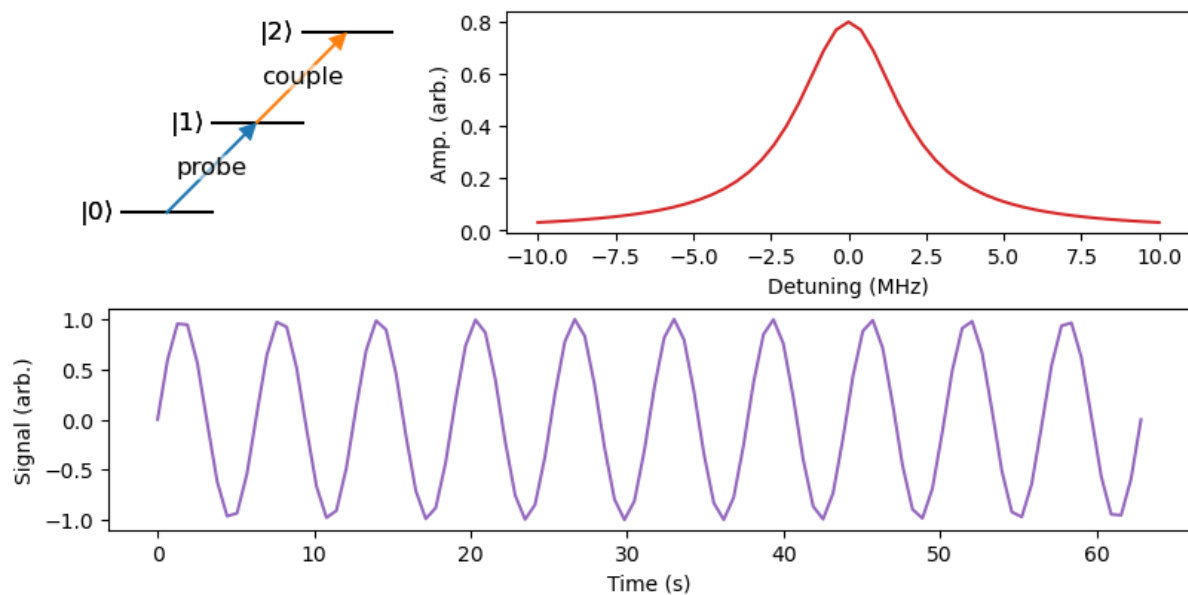
        coupling_defaults={'arrowsize':0.2})
d.draw()
axs['a'].set_aspect('equal')

### part b
axs['b'].plot(bx_data, by_data, color='C3')
axs['b'].set_xlabel('Detuning (MHz)')
axs['b'].set_ylabel('Amp. (arb.)')

### part c
axs['c'].plot(cx_data, cy_data, color='C4')
axs['c'].set_xlabel('Time (s)')
axs['c'].set_ylabel('Signal (arb.)')

```

```
Text(0, 0.5, 'Signal (arb.)')
```



### 1.5.5 Draw graphs using other conventions

```
import rydiqule as rq
```

```

s = rq.Sensor(3)

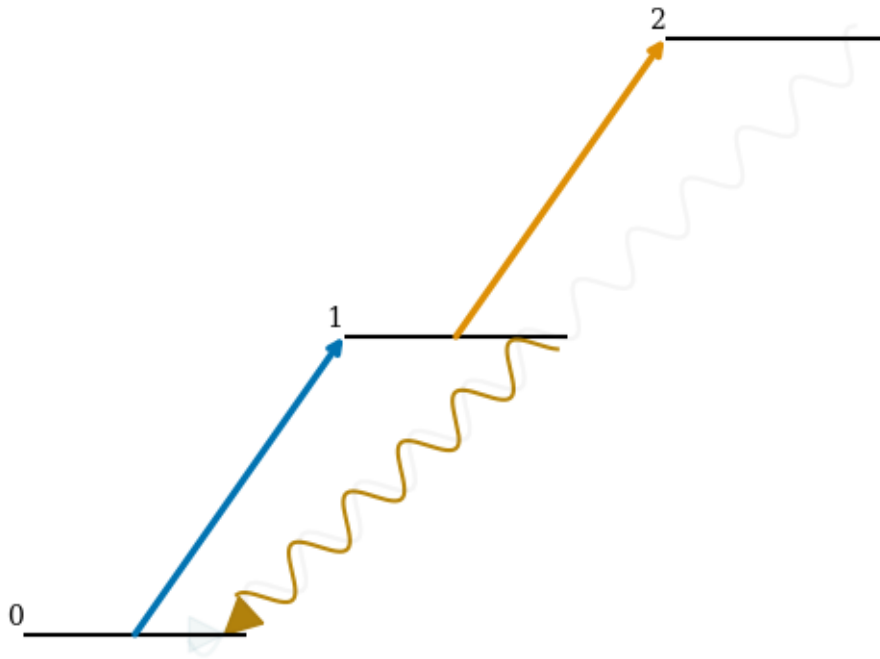
probe = {'states':(0,1), 'rabi_frequency':1, 'detuning':0}
couple = {'states':(1,2), 'rabi_frequency':6, 'detuning': -1}

s.add_couplings(probe, couple)
s.add_decoherence((1,0), 6)
s.add_transit_broadening(0.1)

```

```
rq.draw_diagram(s)
```

```
<rydiqule.energy_diagram.ED at 0x1b8eefbbf70>
```



```
def ryd_to_ld(sensor):

    rq_g = sensor.couplings.copy()

    # level settings
    if isinstance(sensor, rq.Cell):
        for lev, vals in rq_g.nodes.items():
            ld_kw = {}

    # coupling settings
    for edge, vals in rq_g.edges.items():
        ld_kw = {}
        if 'dipole_moment' in vals:
            ld_kw['linestyle'] = 'dashed'
        elif 'rabi_frequency' in vals:
            if not np.all(vals.get('rabi_frequency')):
                ld_kw['hidden'] = True

    # decoherence settings

    # get decoherence normalizations
    gamma_matrix = sensor.decoherence_matrix()
    # we get the biggest possible decoherence value for each term
    # by doing a max reduction along stack axes
    stack_axes = tuple(np.arange(0, gamma_matrix.ndim-2))
    gamma_matrix = gamma_matrix.max(axis=stack_axes)
    max_dephase = gamma_matrix.max()
    min_dephase = gamma_matrix[gamma_matrix != 0.0].min()
    if np.isclose(min_dephase, max_dephase):
        # all non-zero dephasings are the same, prevent /0 error in normalization
        min_dephase = max_dephase*1e-1

    # reversing order of traverse to prevent transit overlaps
    idxs = np.argwhere(gamma_matrix != 0.0)[::-1,:]
```

(continues on next page)

(continued from previous page)

```
for idx in idxs:

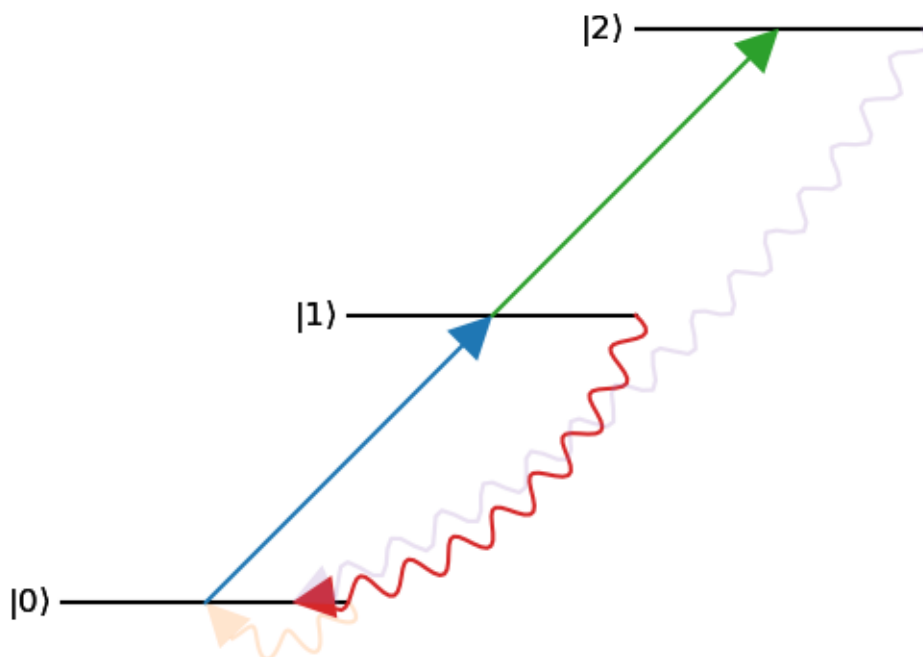
    ld_kw = {}

    ld_kw['wavy'] = True
    ld_kw['deflect'] = True
    ld_kw['start_anchor'] = 'right'
    if idx[0] == idx[1]:
        ld_kw['deflection'] = 0.15
    else:
        ld_kw['stop_anchor'] = (0.3, 0.0)
    # ensure alpha doesn't get too small to not be seen
    # also uses a log scale for the full range of non-zero dephasings
    alph = 1-(0.8*np.log10(gamma_matrix[tuple(idx)]/max_dephase
                          )/np.log10(min_dephase/max_dephase))
    ld_kw['alpha'] = alph

    rq_g.edges[tuple(idx)]['ld_kw'] = ld_kw

return rq_g
```

```
rq_g = ryd_to_ld(s)
d = ld.LD(rq_g, use_ld_kw=True)
d.draw()
```



```
ld.about()
```

```
leveldiagram
=====

leveldiagram Version: 0.2.0
```

(continues on next page)



(continued from previous page)

### Dependencies

=====

```

Python Version:      3.10.8
NumPy Version:       1.23.4
Matplotlib Version:  3.5.3
NetworkX Version:    2.8.4
  
```



## PYTHON MODULE INDEX

|  
leveldiagram.artists, [10](#)  
leveldiagram.utils, [13](#)



## Symbols

`_coupling_defaults` (*leveldiagram.LD attribute*), 9  
`_deflection_defaults` (*leveldiagram.LD attribute*), 9  
`_level_defaults` (*leveldiagram.LD attribute*), 9  
`_wavy_defaults` (*leveldiagram.LD attribute*), 9

## A

`about()` (*in module leveldiagram.utils*), 13

## B

`bra_str()` (*in module leveldiagram.utils*), 13

## C

`Coupling` (*class in leveldiagram.artists*), 10  
`couplings` (*leveldiagram.LD attribute*), 9

## D

`deep_update()` (*in module leveldiagram.utils*), 14  
`draw()` (*leveldiagram.artists.Coupling method*), 11  
`draw()` (*leveldiagram.artists.EnergyLevel method*), 12  
`draw()` (*leveldiagram.LD method*), 9

## E

`EnergyLevel` (*class in leveldiagram.artists*), 12

## G

`generate_couplings()` (*leveldiagram.LD method*), 9  
`generate_levels()` (*leveldiagram.LD method*), 9  
`get_anchor()` (*leveldiagram.artists.EnergyLevel method*), 12  
`get_center()` (*leveldiagram.artists.EnergyLevel method*), 12  
`get_left()` (*leveldiagram.artists.EnergyLevel method*), 13  
`get_right()` (*leveldiagram.artists.EnergyLevel method*), 13

## I

`init_arrowheads()` (*leveldiagram.artists.Coupling method*), 11  
`init_label()` (*leveldiagram.artists.Coupling method*), 11  
`init_path()` (*leveldiagram.artists.Coupling method*), 11

## K

`ket_str()` (*in module leveldiagram.utils*), 14

## L

`LD` (*class in leveldiagram*), 8  
`leveldiagram.artists`  
    *module*, 10  
`leveldiagram.utils`  
    *module*, 13  
`levels` (*leveldiagram.LD attribute*), 10

## M

*module*  
    `leveldiagram.artists`, 10  
    `leveldiagram.utils`, 13

## S

`set_axes()` (*leveldiagram.artists.Coupling method*), 11  
`set_axes()` (*leveldiagram.artists.EnergyLevel method*), 13  
`set_data()` (*leveldiagram.artists.EnergyLevel method*), 13  
`set_figure()` (*leveldiagram.artists.Coupling method*), 11  
`set_figure()` (*leveldiagram.artists.EnergyLevel method*), 13  
`set_transform()` (*leveldiagram.artists.Coupling method*), 12  
`set_transform()` (*leveldiagram.artists.EnergyLevel method*), 13

## T

`text_labels` (*leveldiagram.artists.EnergyLevel attribute*), 13