rydiqule

Release 2.1.3

Quantum Technology Center
DEVCOM Army Research Laboratory
Naval Air Warfare Center - Weapons Division

Distribution Statement A - Approved for public release: distribution is unlimited

Mr. Benjamin Miller

Dr. David H Meyer

Dr. Kevin C Cox

Dr. Christopher O'brien

Mr. Teemu Virtanen

Apr 16, 2026

GETTING STARTED

1 Installation 3
- 1.1 Regular Installation . . . 3
- 1.2 Editable Installation . . . 4
- 1.3 Confirm installation . . . 5
- 1.4 Updating an existing installation . . . 5
- 1.5 Dependencies . . . 6

2 Rydiqule Overview 7
- 2.1 Define the System . . . 7
- 2.2 Solve the System . . . 8
- 2.3 Interpret the Results . . . 8

3 Introduction to Rydiqule 9
- 3.1 Design philosophy . . . 9
- 3.2 Limitations . . . 10
- 3.3 1. The `Sensor` object . . . 10
- 3.4 2. Solving a `Sensor` in the steady-state case . . . 19
- 3.5 3. Solving in the time domain . . . 30
- 3.6 4. Simulating Doppler broadening . . . 43
- 3.7 5. What Next? . . . 45

4 Introduction to `Cell` and Real Atoms 47
- 4.1 0. What is a `Cell`? . . . 47
- 4.2 1. Creating a `Cell` . . . 47
- 4.3 2. Decoherence rates in `Cell` . . . 51
- 4.4 3. Couplings in `Cell` . . . 56
- 4.5 4. Solving systems defined in `Cell` . . . 60

5 Rydiqule Performance 63
- 5.1 Steady-state solve time scaling versus basis size . . . 63
- 5.2 Steady-state solve time scaling versus stack size . . . 65
- 5.3 Time-dependent solve times versus stack size . . . 66
- 5.4 Comparison to qutip . . . 68

6 Changelog 77
- 6.1 v2.1.3 . . . 77
- 6.2 v2.1.2 . . . 77
- 6.3 v2.1.1 . . . 78
- 6.4 v2.1.0 . . . 78
- 6.5 v2.0.0 . . . 78
- 6.6 v1.2.3 . . . 81
- 6.7 v1.2.2 . . . 81
- 6.8 v1.2.1 . . . 81
- 6.9 v1.2.0 . . . 81
- 6.10 v1.1.0 . . . 82

6.11 v1.0.0 . . . 83
6.12 v1.0.0rc2 . . . 83
6.13 v1.0.0rc1 . . . 84
6.14 v0.5.0 . . . 84
6.15 v0.4.0 . . . 85
6.16 v0.3.0 . . . 86
6.17 v0.2.0 . . . 87
6.18 v0.1.0 . . . 88

7 Physics Documentation **89**
7.1 Equations of Motion Generation . . . 89
7.2 Stacking Conventions . . . 92
7.3 Observables . . . 93
7.4 Doppler Averaging . . . 94
7.5 Time-Dependence . . . 98
7.6 Solving with Atomic Structure . . . 103

8 API Documentation **111**
8.1 rydiqule . . . 111

9 Developer Documentation **261**
9.1 Unit Tests . . . 261
9.2 Type Hinting . . . 262
9.3 Linting . . . 263
9.4 Building the Documentation . . . 263
9.5 Contributing . . . 264

10 3-photon Rydberg EIT **267**
10.1 Doppler-free, 3 photon excitation . . . 267
10.2 Colinear 3-photon Excitation with Doppler Averaging . . . 268
10.3 Doppler-Free Angles . . . 270

11 Analytic Doppler Solver **273**
11.1 1D Doppler Averaging . . . 273
11.2 2D Doppler Averaging . . . 276
11.3 3D Doppler Averaging . . . 281

12 Calculating SNR **287**
12.1 1D Optimum . . . 287
12.2 2D Optimum - Fnd Optimized Ω_p and Ω_c for best SNR . . . 289

13 Simple NMOR examples **293**
13.1 A F=0 to F'=1 spectroscopy experiment with $\sigma^{\pm}$ probing light . . . 293
13.2 A F=0 to F'=1 spectroscopy experiment with π polarized probe light . . . 295
13.3 A F=1 to F'=0 spectroscopy experiment with $\sigma^{\pm}$ and a static axial magnetic field . . . 296
13.4 A F=1 to F'=0 spectroscopy experiment with $\sigma^{\pm}$ and a static detuning and a varying axial magnetic field . . . 298
13.5 A Rb87 D2 spectroscopy experiment with $\sigma^{\pm}$ polarized light . . . 300

14 RF heterodyne with Doppler Example **305**
14.1 Imports . . . 305
14.2 Define the Sensors . . . 305
14.3 Observe a heterodyne beat between the Signal and LO. . . . 306

15 RF heterodyne example **311**
15.1 Imports . . . 311
15.2 Comparing the steady-state and time solver results . . . 311
15.3 Observe a heterodyne beat between the Signal and LO. . . . 314
15.4 RF Heterodyne in the Rotating Wave Approximation . . . 317
15.5 Find the optimum laser detuning with LO . . . 318

15.6 Test the Linear Dynamic Range 319

16 Modeling Saturated Absorption Spectroscopy **323**

Python Module Index **331**

A python library for calculating Rydberg electrometer response to arbitrary RF fields in steady-state or time domains. It is a general density matrix-based master equation solver, optimized for speed to solve problems with large parameter spaces while maintaining flexibility to define novel problems. It leverages a graph-based system definition, computationally-efficient equation "stacking" in the form of tensors, and external computational libraries such as `numpy`, `scipy`, and `ARC`.

For more details, see the *Rydiqule Overview*.

For detailed usage examples, see the *Introduction to Rydiqule* Jupyter notebook.

If you use rydiqule in your work, please cite as

```
@article{rydiqule_2024,
   author = {Miller, B. N. and Meyer, D. H. and Virtanen, T. and O'Brien, C. M.␣
↪and Cox, K. C.},
   title = {RydIQule: A Graph-based paradigm for modeling Rydberg and atomic␣
↪sensors},
   journal = {Computer Physics Communications},
   volume = {294},
   pages = {108952},
   year = {2024},
   doi = {10.1016/j.cpc.2023.108952},
   url = {https://doi.org/10.1016/j.cpc.2023.108952},
   eprint = {https://doi.org/10.1016/j.cpc.2023.108952}
}
```

CHAPTER

ONE

INSTALLATION

Installation is done via pip or conda. See below for detailed instructions.

In all cases, it is highly recommended to install rydiqule in a virtual environment.

Installation via conda is recommended for rydiqule. It handles dependency installation as well as a virtual environment to ensure packages do not conflict with other usages on the same system. Finally, the `numpy` provided by anaconda has been compiled against optimized BLAS/LAPACK implementations, which results in much better performance in rydiqule itself.

Note

RydIQule currently requires python >3.8. For a new installation, it is recommended to use the newest supported python. Currently supported versions are

1.1 Regular Installation

Assuming you have not already created a separate environment for RydIQule (recommended), run the following to create a new environment:

```
(base) ~/> conda create -n rydiqule python=3.11
(base) ~/> conda activate rydiqule
```

Now install via rydiqule's anaconda channel. This channel provides rydiqule as well as its dependencies that are not available in the default anaconda channel. If one of these dependencies is outdated, please raise an issue with the vendoring repository.

```
(rydiqule) ~/> conda install -c rydiqule rydiqule
```

To install normally, run:

```
pip install rydiqule
```

This command will use pip to install all necessary dependencies.

The `uv` package and project manager allows for unified project management in a way that is reproducible and easy to share. You create a new simulation project that relies on `rydiqule` by running the following commands.

```
uv init new-project
cd new-project
uv venv --python 3.12  # this line optionally sets python version to use for the␣
↪venv
```

These commands create a project template in the sub-directory `new-project`. This directory is readily version controlled and contains all information needed to reproduce your environment.

You finish configuring the project by adding required dependencies, starting with `rydiqule`.

```
uv add rydiqule
```

You can now create a script in the project and run it directly:

```
uv run example.py
```

You can also add jupyter kernel support and run jupyter notebooks from VS Code. This support is added as a development dependency.

```
uv add --dev ipykernel
```

Once added, VS Code can launch a jupyter notebook using the project's virtual environment located in the `/new-project/.venv/` sub-directory.

You can also run jupyter lab directly from the virtual environment with the following command

```
uv run --with jupyter jupyter lab
```

1.2 Editable Installation

If you would like to install rydiqule in editable mode to locally modify its source, use the following commands.

Follow the above to install rydiqule and its dependencies, then run the following to uninstall rydiqule as provided by conda and install the editable local repository.

```
(rydiqule) ~/> conda remove rydiqule --force
# following must be run from root of local repository
(rydiqule) ~/> pip install -e .
```

Run the following from the root directory of the cloned repository:

```
pip install -e .
```

Using `uv` on an existing python package automatically installs it in editable mode. Run the following from the root of the local repository.

```
uv venv --python 3.12  # this line optionally sets a python version for the venv
uv sync
```

Note that `uv sync` automatically installs the `dev` dependency group which includes `ipykernel` and `pytest`.

You can now use VS Code to run jupyter notebooks with the virtual environment at `rydiqule/.venv/`. To use jupyter lab, run the following command

```
uv run --with jupyter jupyter lab
```

Note that editable installations should have `git` available if you want dynamic versioning (via `setuptools-scm`), either by a system-wide installation or via conda in the virtual environment (`conda install git`).

Note

While rydiqule is a pure python package (ie it is platform independent), its core dependency ARC is not. If a pre-built package of ARC is not available for your platform in our anaconda channel, you will need to install ARC via `pip` to build it locally before installing `rydiqule`. To see what architectures are supported, please see the vendoring repository.

1.3 Confirm installation

Proper installation can be confirmed by executing the following commands in a python terminal.

```
>>> import rydiqule as rq
>>> rq.about()

        Rydiqule
    ================

Rydiqule Version:     1.1.0
Installation Path:    ~\Miniconda3\envs\rydiqule\lib\site-packages\rydiqule

      Dependencies
    ================

NumPy Version:        1.24.3
SciPy Version:        1.10.1
Matplotlib Version:   3.7.1
ARC Version:          3.3.0
Python Version:       3.9.16
Python Install Path:  ~\Miniconda3\envs\rydiqule
Platform Info:        Windows (AMD64)
CPU Count:            12
Total System Memory:  128 GB
```

1.4 Updating an existing installation

Upgrading an existing installation is simple. Simply run the appropriate upgrade command for the installation method used.

1.4.1 Regular Installation Upgrade

```
conda upgrade rydiqule
```

```
# standard upgrade
pip install rydiqule
# greedy upgrade: ie update dependencies too
pip install -U rydiqule
```

```
uv lock --upgrade-package rydiqule
# greedy upgrade
uv lock --upgrade
```

1.4.2 Editable Installation Upgrade

If using an editable install, simply replacing the files in the same directory is sufficient. Though it is recommended to also run the appropriate pip update command as well to capture updated dependencies.

```
pip install -U -e .
```

```
pip install -U -e .
```

```
uv sync
```

Note that any `uv run` command will automatically sync and thereby capture updated dependencies.

1.5 Dependencies

This package requires installation of the excellent ARC package, which is used to get Rydberg atomic properties. It also requires other standard computation dependencies, such as `numpy`, `scipy`, `matplotlib`, etc. These will be automatically installed if not already present.

Note

Rydiqule's performance does depend on these dependencies. In particular, `numpy` can be compiled with a variety of backends that implement BLAS and LAPACK routines that can have different performance for different computer architectures. When using Windows, it is recommended to install `numpy` from the conda default channel, which is built against the IntelMKL and has generally shown the best performance for Intel-based PCs.

Optional timesolver backend dependencies include the `numba` and CyRK packages. Both are available via `pip`, `conda`, or our anaconda channel.

For conda installations, these dependencies must be installed manually

```
conda install -c rydiqule CyRK
```

Backends can be installed automatically via the optional extras specification for the `pip` command.

```
pip install rydiqule[backends]
```

Backends can be installed automatically via the optional extras specification for the `uv sync` command.

```
uv sync --extra backends
```

Note that these dependencies will be uninstalled if `uv sync` is called without the extras flag.

CHAPTER TWO

RYDIQULE OVERVIEW

Rydiqule (RYDberg sensing Interactive Quantum ModULE) is a python module that can calculate response of a Rydberg sensor to RF fields. It uses a semi-classical approximation of the Schroedinger equation known as the Lindblad equation to create equations of motion that describe the interaction of the sensor with optical and RF fields.

In order to use rydiqule at its basic level, you need to understand a few core elements. These elements are shown in Fig. 2.1.

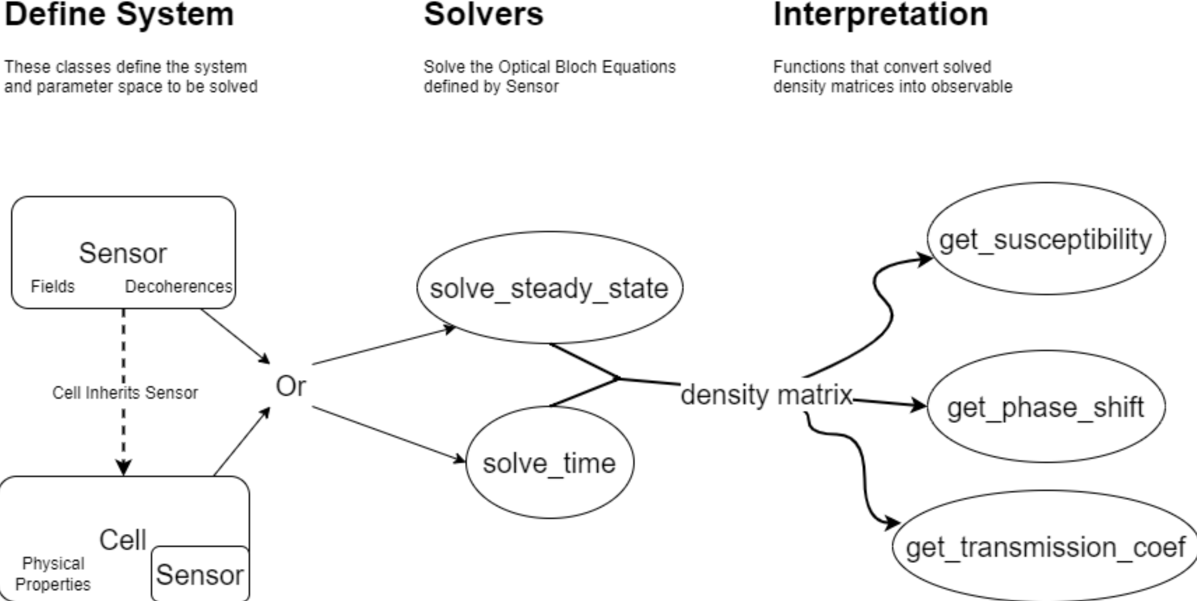


Fig. 2.1: The basic organizational structure for using Rydiqule.

A calculation needs three general components.

1. Define the system to be solved.
2. Solve the system.
3. Interpret the results to observable quantities.

2.1 Define the System

A system can be defined using one of two classes: `Sensor` or `Cell`. In either case, multiple values for a parameter can be set to produce parameter scans in the solve stage. The Sensor class defines the bare minimum of information necessary to produce a system of equations to solve. This class allows for arbitrary definitions of the system. The Cell class defines a physical gas of atoms for the system that in turn produces a Sensor for doing calculations. This class relies on ARC to provide physical parameters.

2.2 Solve the System

We currently have three solvers implemented.

1. A steady state solver (`solve_steady_state()`) that finds the steady state density matrix of the system. This can solve in a few different conditions:

   1. Optically-thin, cold ensemble

   2. Optically-thin, doppler-averaged ensemble

2. **A steady state doppler solver (`solve_doppler_analytic()`) that analytically finds the steady state density**
   matrix of an optically-thin, doppler-averaged ensemble.

3. A time solver (`solve_time()`) that allows for fields to be defined arbitrarily in time.

Each solver takes a Sensor or Cell object and solves the system. The output is the corresponding density matrix in the steady-state, spanning the defined parameter space, or a series of density matrices versus time in the case of the time solver.

2.3 Interpret the Results

Once the solultions are made, we need to interpret the density matrices into observable values, typically some change to the optical probing field. We implement a few functions to get the `susceptibility`, `probe absorption`, and `probe phase shift`. Note that getting these values generally requires more information about the system than the bare minimum required to solve them. *Raw density matrix elements* can also be obtained.

CHAPTER

THREE

INTRODUCTION TO RYDIQULE

RydIQule, the Rydberg Interactive Quantum Module, is a python library built to simulate the interaction of Rydberg atoms and light using a semi-classical approach. This notebook will illustrate some of its core functionality, and demonstrate how to use the tool to model simple systems. The intent here is not to demonstrate discoveries in physics, or even simulate useful physics generally. The values provided are deliberately simple integers to make the functions as easy to understand as possible. Thus, the notebook will use cartoonish but nontrivial (from a code perspective) examples to demonstrate how to use the module. Feel free to run the cells of this notebook as-is, but also by all means add and modify cells to get a better feel for how the package works.

This notebook can be downloaded here.

The functionality and principles behind `rydiqule` are detailed much more thoroughly in the documentation. A few places in the docs are worth pointing out:

1. API Documentation contains detailed information about how the functions of rydiqule behave from a software perspective.
2. Physics Documentation contains writeups detailing the physics and math conventions used by rydiqule. There are places where `rydiqule` diverges from the conventions of atomic physicists for numerical or code-related reasons, or makes explicit decisions about conventions that are not universal. It is a very helpful reference when something is not working as you expect.
3. There are example notebooks that can be accessed from the docs homepage which contain more realistic use cases for rydiqule.

3.1 Design philosophy

Rydiqule was built with a few core principles in mind:

1. **Rydiqule is simple** - Setting and solving an atomic system can be done with just a handful of lines of code while behaving in an intuitive way.
2. **Rydiqule is fast** - Under the hood, the library makes broad use of fast numpy matrix broadcasting and compiled code in places that would be slowed down by native python. The result is a toolbox that can produce meaningful results in a few minutes or less. This is especially true for steady-state solutions which can be extremely fast, even for large parameter spaces. With version 2.2.0, we introduced a fully compiled time-solver backend that significantly improved time-dependent solve times, bringing them more in line with what is fundamentally achievable.
3. **Rydiqule is flexible** - Rydiqule can model a huge variety of semiclassical Rydberg atomic systems with no code modification. For users with more particular modelling needs who wish to extend Rydiqule, the `Sensor` class provides a minimal physical system that can easily be inherited and overloaded for more involved experimental setups (such as `Cell`, which provides an ARC-integrated wrapper to `Sensor` for calculating Rydberg physics using atomic properties and lab-measured parameters).

3.2 Limitations

While we have worked hard to make Rydiqule as good as possible, there are some areas that can cause issues:

1. **Memory** - `rydiqule`'s speed primarily comes at the cost of increased memory footprint. In order to avoid Python interpreter overhead and fully leverage the speed of Numpy broadcasting and C-level for-looping when solving multiple sets of parameters, we have to pass all equations of motion as a single tensor object to the low-level C functions (such as those backing `numpy.linalg.solve` used by the steady-state solver). For systems with many laser parameter values, many levels, doppler averaging in several dimensions, or especially a combination of these, the equations of motion generated by rydiqule are very large, often requiring more memory than is in a typical laptop or simple desktop. For very large systems, the memory footprint may even outpace a powerful workstation. Rydiqule has built-in functionality to handle this, but it is not fully general and it will lead to slower solves.
2. **Speed** - While huge improvements have been made in the speed of `rydiqule` since its alpha stages, there are certain situations that can still cause slowdowns. For longer time-dependent simulations with large detunings, in particular for the poorly-conditioned equations produced with large doppler width, solving can still be slow. With version 2.2.0, we introduced a fully compiled time-solver backend that greatly improves performance in these situations, but the fundamental time-dependence from large detunings is still present, bounding time-dependent solver performance relative to steady-state solving. Further improvements can be gained from algorithmic developments (such as Floquet techniques), but they typically enforce constraints on the time-dependence that can be modeled.
3. **Quantum Back-action** - We treat the optical fields as static, and do not include them explicitly in the semi-classical equations of motion. Rydiqule does not account for atom-field back-action effects. This approximation is valid for low optical depth samples, and is known to give valid results for SNR in moderate optical depth samples. However, for quantitative analysis of quantum noise in high optical depth samples, Rydiqule may not be accurate. By extension, rydiqule is also not presently suitable for simulations involving single-photon-level fields.
4. **Device Modelling** - Rydiqule is a physics solver, and does not currently have user-friendly support for device-level modelling. Put another way, rydiqule requires a certain level of atomic physics understanding to ensure results are physically meaningful/correct.

3.2.1 Imports

Rydiqule is conventionally imported as `rq`. In addition, `numpy` and `matplotlib` are almost always useful to have in notebooks using rydiqule, so we will import them as well. They are dependencies of `rydiqule`, so they should already be installed if you installed rydiqule.

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

3.3 1. The `Sensor` object

The `Sensor` is the core object of rydiqule. It defines an atomic system, and while you can create and solve a Hamiltonian manually, the `Sensor` takes care of the bookeeping to generate Hamiltonians and solve its associated equations of motion.

Note: Don't think too hard about why it is called a `Sensor`, it is a legacy name that does not necessarily capture its entire functionality.

3.3.1 1.1 The quantum state basis of `Sensor`

The base `Sensor` class is constructed with a single required argument, which specifies which states are in the basis. Here we create a 3-level system in the manner `rydiqule` supports: an integer defining the basis size. On its own, a `Sensor` contains no information about atomic structure; it is an abstraction that allows for a high degree of manual configuration.

```
basis_size = 3
simple_sensor = rq.Sensor(basis_size)
```

This is the simplest way to define a `Sensor` with a particular number of states (3 in this case). In this case, the basis states will simply be labeled `[0,1,2]`, ($|0\rangle, |1\rangle, |2\rangle$). However, mutiple other ways exist to label the basis states. Rather than an integer (which will implcitly use python's `range()`), we can pass a list of basis state labels directly. These labels can either be strings or tuples. Tuples in particular can be useful for representing quantum numbers, although they can represent anything that you think is useful; they do not invoke any physics in `Sensor`.

```
string_sensor = rq.Sensor(['g', 'e1', 'e2'])
tuple_sensor = rq.Sensor([(0,0), (1,-1), (1,0), (1,1)])
```

Once we have created a `Sensor`, we can use the `Sensor.states` attribute to see the states. Note that the ordering of states in a `Sensor` will always be as they are provided in the constructor.

```
print(simple_sensor.states)
print(string_sensor.states)
print(tuple_sensor.states)
```

```
[0, 1, 2]
['g', 'e1', 'e2']
[(0, 0), (1, -1), (1, 0), (1, 1)]
```

Specifications of tuples like the one above can become cumbersome, especially to represent large groups of states. For tuples, `rydiqule` also supports "state specifications", in which one of the tuple elements is provided as a list. `rydiqule` will automatically expand such a state into a list as demonstrated below. Note that even when a specification like this is used, it is still necessary to enclose it in list brackets in the constructor. Furthermore, it is often useful to define tuples outside functions, since the number of brackets and parentheses can become cumbersome. This is a safe operation since tuples in python are immutable.

```
g = (0,0)
e = (1, [-1,0,1])
state_spec_sensor = rq.Sensor([g,e])
print(state_spec_sensor.states)
```

```
[(0, 0), (1, -1), (1, 0), (1, 1)]
```

3.3.2 1.2 `Sensor` Data structure: The Graph

Before we proceed to useful physics, it is worth a brief sidebar to discuss how the information about levels and couplings is stored in a `Sensor`. NetworkX is a library for storing and analyzing graphs in python that uses dictionaries to store information about nodes and edges. At the core of a `Sensor` is a `networkx` graph. The energy levels in an atomic system are stored as nodes and the couplings between states are stored as edges. Above, when we define a "3-level system", what `rydiqule` has created under the hood is a `networkx` directed graph with 3 nodes and no edges. We can access this graph to view information about it with the `Sensor.couplings` attribute. Let's inspect the `tuple_sensor` we made above and see that the graph nodes are indeed the states we printed above.

```
print(tuple_sensor.couplings)
print(tuple_sensor.couplings.nodes)
```

```
DiGraph with 4 nodes and 0 edges
[(0, 0), (1, -1), (1, 0), (1, 1)]
```

Indeed the `states` attribute is just a wrapper for `couplings.nodes`. Note that this is provided only as an interface to view the graph, and it shoud **NOT** be modified directly, as doing so may cause `rydiqule` to either `Error` or not work as expected.

3.3.3 1.3 Coupling states with simple electromagnetic fields:

While its great to create basis states, there isn't any physics in an atomic system of completely uncoupled basis states. Fortuntely, it is easy to couple states together in `rydiqule` to start modelling interesing physics. States are coupled together using, at minimum, the following:

1. A 2-element tuple representing the states to be coupled.
2. The rabi_frequency. `rydiqule` treats all frequencies in Mrad/s.
3. The detuning (for couplings to be treated in the rotating frame), or the transition frequency.

Notes on detunings and rotating wave transformation

When we define detunings, the states in a coupling are always defined to go from lower to higher energy. For example, `(1,2)` means state 2 is higher than 1. Any coupling with a defined `detuning` will be treated in a rotating frame, with positive detuning always treated as a blue detuning. For now, we will only consider detunings, but later we will discuss transition frequency specification in the cotext of exactly modelling time-dependant effects.

Here we will demonstrate defining a few different systems, showing different equivalent methods for each.

Defining a simple Ladder system

This demonstrates how a minimal ladder system with simple values might be defined in `rydiqule`. Here we use the standard integer basis and the `add_couplings` method.

```
ladder_sensor = rq.Sensor(3)
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
ladder_sensor.add_couplings(laser_01, laser_12)
```

To demonstrate what happens under the hood, let us inspect the `Sensor.coupling` object again, this time looking at the graph edges. We can look at just `edges`, for an overview of which nodes are coupled, or we can use `edges(data=...)` for more thorough representations.

```
print("Edge Keys: ", ladder_sensor.couplings.edges)
print("Edge Data: ", ladder_sensor.couplings.edges(data=True))
print("Edge Rabi: ", ladder_sensor.couplings.edges(data="rabi_frequency"))
```

```
Edge Keys:  [(0, 1), (1, 2)]
Edge Data:  [(0, 1, {'rabi_frequency': 3, 'detuning': 1, 'phase': 0, 'kvec': (0, 0,
↪ 0), 'coherent_cc': 1.0, 'label': '(0,1)'}), (1, 2, {'rabi_frequency': 5,
↪'detuning': 2, 'phase': 0, 'kvec': (0, 0, 0), 'coherent_cc': 1.0, 'label': '(1,2)
↪'})]
Edge Rabi:  [(0, 1, 3), (1, 2, 5)]
```

As we can see, the options described above are just the minimum; there are more keywords we can use to describe a coupling between two states. More complicated uses will be discussed in detail in other places in this notebook, but here is a brief overview of each optimal argument:

1. `phase` describes the relative phase of a field, and will scale the corresponding off-diagonal terms with an $e^{i\theta}$ phase offset.
2. `kvec` is a description of a laser's k-vector, and is described by (x,y,z) components. This is relevant when we discuss doppler-broadened calculations later. For now, leaving it at the default will ignore doppler effects.
3. `coherent_cc` is a general abstraction of what is essentially a Clebsch-Gordon coefficient. These will become relevant when we discuss coupling manifolds of states later. The default is 1, and that will effectively ignore the value.
4. `label` is a string shorthand that can reference the coupling in some other contexts. Useful labels might include `"blue"`, `"red"`, `"probe"`, `"pumping"`. This label will enforce nothing with regards to physics, but may be helpful. If it is not supplied, it will default to a string cast of the tuple describing the states, as above.

Any of these values can be defined in the coupling in the exact same manner as "detuning" or "rabi_frequency".

Printing `Sensor` Data

Since calling that info above is a little bit tedious, `rydiqule` also supports directly printing a sensor using python's `print()` function, which will summarize basic information about a `Sensor`. We will typically use this method of displaying information about the graph in these notebooks, since it prints in a more readable way.

```
print(ladder_sensor)
```

```
<class 'rydiqule.sensor.Sensor'> object with 3 states and 2 coherent couplings.
States: [0, 1, 2]
Coherent Couplings:
    (0,1): {rabi_frequency: 3, detuning: 1, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 5, detuning: 2, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

We will discuss Decoherent Couplings and Energy shifts, later, but for now we can see this is a substantially easier way to view information about the `Sensor`

Automatic Hamiltonian generation

Once the system is defined, we can see the Hamiltonian matrix. In many ways, this is the core piece of `rydiqule` that makes it useful. You usually will not need to call this explicitly (it will be called internally by a solver, more on that later), but it can be useful to make sure the system is defined as expected.

```
print(ladder_sensor.get_hamiltonian())
```

```
[[ 0. +0.j  1.5+0.j  0. +0.j]
 [ 1.5-0.j -1. +0.j  2.5+0.j]
 [ 0. +0.j  2.5-0.j -3. +0.j]]
```

Note the detuning sign convention. We have added a positive detuning value to the system. `rydiqule` chooses a rotating frame automatically, and there is no user-configurable option to change it. The roating frame chosen is one in which a detuning of 1 corresponds to an term of -1 on the Hamiltonian diagonal.

Defining a simple Vee scheme

We can do something similar, but with a different arrangement of couplings, and using explicit strings to define the basis. We can also pass fields in the constructor if we like.

```
basis = ["g", "e1", "e2"]
laser_01 = {"states": ("g","e1"), "detuning": 1, "rabi_frequency": 3, "label":"red
↪"}
laser_02 = {"states": ("g","e2"), "detuning": 2, "rabi_frequency": 5, "label":"blue
↪"}
sensor_v = rq.Sensor(basis, laser_01, laser_02)
print(sensor_v.couplings.edges(data="label"))
print(sensor_v.get_hamiltonian())
```

```
[('g', 'e1', 'red'), ('g', 'e2', 'blue')]
[[ 0. +0.j  1.5+0.j  2.5+0.j]
```

(continues on next page)

(continued from previous page)

```
 [ 1.5-0.j -1. +0.j  0. +0.j]
 [ 2.5-0.j  0. +0.j -2. +0.j]]
```

Defining a simple Lambda scheme

Here is another system, this time a lambda scheme. Here we demonstrate another way to add couplings using the `add_coupling` function. This function behaves equivalently to `add_couplings`, and which you choose is up to preference. As one final way of defing our basis, let's use tuples here. This can be a little more involved, but occasionally useful in large sytems. As discussed above, we will alias the states so couplings don't start getting clunky to define. We also show the `expand_statespec` function to turn the `e` variable into a list of all possible values.

```
g = (0,0)
e = (1, [-1,0,1])
print("e states: ", rq.expand_statespec(e)) #just creates a list, no physics is␣
↪done here
[e1, e2, e3] = rq.expand_statespec(e) #unpack the list for easy use later.

sensor_lambda = rq.Sensor([g, e]) #note that ryqiqule will automatically unpack e␣
↪into a list

sensor_lambda.add_coupling((g, e1), rabi_frequency=1, detuning=1)
sensor_lambda.add_coupling((e1,e2), rabi_frequency=2, detuning=2)
sensor_lambda.add_coupling((e3,e2), rabi_frequency=3, detuning=3) #tuples always␣
↪go from low to high energy by convention

print(sensor_lambda)
print(sensor_lambda.get_hamiltonian())
```

```
e states:  [(1, -1), (1, 0), (1, 1)]
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: ((0, 0),(1, -1))}
    ((1, -1),(1, 0)): {rabi_frequency: 2, detuning: 2, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: ((1, -1),(1, 0))}
    ((1, 1),(1, 0)): {rabi_frequency: 3, detuning: 3, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: ((1, 1),(1, 0))}
Decoherent Couplings:
    None
Energy Shifts:
    None
[[ 0. +0.j  0.5+0.j  0. +0.j  0. +0.j]
 [ 0.5-0.j -1. +0.j  1. +0.j  0. +0.j]
 [ 0. +0.j  1. -0.j -3. +0.j  1.5-0.j]
 [ 0. +0.j  0. +0.j  1.5+0.j  0. +0.j]]
```

Systems that are not fully coupled

We can also define a system in which not all states are connected explicitly by couplings. This allows us to solve systems which, for example, have states coupled only by decoherence. This exact use case will be demonstrated later, but we can set up the system and show the hamiltonian here.

The hamiltonaian works by treating an uncoupled states as a "second ground state", and calculates digonal hamiltonian elements from there. We show a 4-level system in which state 4 is coupled to state 5 via a steady state rf transition, but to no other states. It should be noted that calling an rf transition does not change the way the system is solved here.

Looking at the hamiltonian, we can see that the 4th term along the diagonal is 0, and the 5th term counts from there.

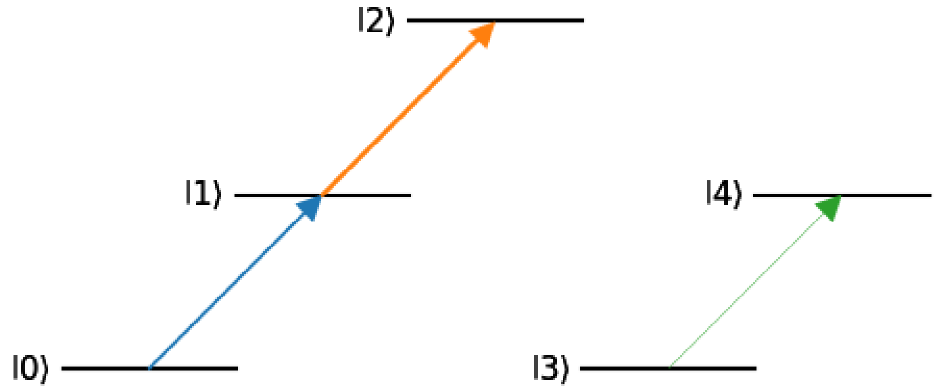
```
sensor_uncoupled = rq.Sensor(5)
laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
rf = {"states": (3,4), "detuning": 8, "rabi_frequency": 1}
sensor_uncoupled.add_couplings(laser_01, laser_12, rf)

print(sensor_uncoupled.get_hamiltonian())
```

```
[[ 0. +0.j  1.5+0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 1.5-0.j -1. +0.j  2.5+0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  2.5-0.j -3. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0. +0.j  0.5+0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0.5-0.j -8. +0.j]]
```

In some more complicated cases, we may want a more concrete visual reassurance that everything is defined correctly. `rydiqule` makes use of the `leveldiagram` library (which uses matplotlib as a backend) to help draw visual representations of a `Sensor`. In the drawing below, we can see that the diagram indeed shows that states 2 and 3 are not coupled. Keep in mind that these drawings have limitations. Especially in very large, highly-coupled systems, the output can get quite cramped, although it is still a helpful sanity check.

```
rq.draw_diagram(sensor_uncoupled)
```

```
<leveldiagram.ld.LD at 0x2333b297dd0>
```



3.3.4 1.5 Coupling state manifolds

So far we have shown the basics of adding couplings. There is one final important detail regard coherent couplings, and that is the use of `add_coupling` to couple entire groups of states with a single call. To do so, we define a `statespec` as in the Lambda example above. Let's construct a similar `Sensor`.

```
g = (0,0)
e = (1, [-1,0,1])
[e1, e2, e3] = rq.expand_statespec(e)

sensor_man = rq.Sensor([g, e])
print(sensor_man.couplings.nodes)
```

```
[(0, 0), (1, -1), (1, 0), (1, 1)]
```

Once we have the `Sensor` defined, we can actually couple the ground state to every excited state with a single call by simply treating `e` as one state. Note that in this case, `"label"` becomes a required argument. Defining in this way

will automagically loop over every state in the `e` "manifold" and add a coupling between `g` and each one. Let's add the coupling and see what edges get added.

```
sensor_man.add_coupling((g,e), rabi_frequency=2, detuning=1, label="probe")
print(sensor_man)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: probe_0, coherent_cc: 1.0}
    ((0, 0),(1, 0)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: probe_1, coherent_cc: 1.0}
    ((0, 0),(1, 1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: probe_2, coherent_cc: 1.0}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

Note that as-is, these 3 states are degenerate. There are a few modifications we can make.

1. We can use the `add_energy_shifts` function to break the degeneracy. This will have the effect of adding diagonal terms to the hamiltonian.
2. Now the `coherent_cc` we saw above becomes relevant. We can add a dictionary that represents a "weighting" to each individual coupling in the group, and pass it under the "coupling_coefficients" keyword. This will not change the rabi_frequency directly, but it will apply a prefactor to the rabi_frequency when generating the hamiltonian.

We will re-make the sensor, and make some sensible choices to demonstate `energy_shifts` and `coupling_coefficients`, in this case using dictionaries. Take a moment to understand this part: it is crucial to understand how to do more advanced calculations. Here we also see where the `Energy Shifts` section of the printout comes in.

```
g = (0,0)
e = (1, [-1,0,1])
[e1, e2, e3] = rq.expand_statespec(e)

#defing coupling coefficients and energy shifts
cc = {
    (g,e1): 0.25,
    (g,e2): 0.5,
    (g,e3): 0.25
}
e_shifts = {e1:-0.1, e3: 0.1}
sensor_man = rq.Sensor([g, e])
sensor_man.add_energy_shifts(e_shifts)
sensor_man.add_coupling((g,e), detuning=1, rabi_frequency=2, coupling_
↪coefficients=cc, label="foo")

print(sensor_man)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_0, coherent_cc: 0.25}
    ((0, 0),(1, 0)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_1, coherent_cc: 0.5}
```

(continues on next page)

(continued from previous page)

```
    ((0, 0),(1, 1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_2, coherent_cc: 0.25}
Decoherent Couplings:
    None
Energy Shifts:
    (1, -1): -0.1
    (1, 1): 0.1
```

One thing to note here is that `e_shift` actually adds an edge from a node to itself. This isn't a crucial detail, but can be helpful to know if you need to debug a complicated system. Now that everything on the graph makes sense, lets see what the Hamiltonian does:

```
print(sensor_man.get_hamiltonian())
```

```
[[ 0.   +0.j   0.25+0.j   0.5 +0.j   0.25+0.j]
 [ 0.25-0.j -1.1 +0.j   0.   +0.j   0.   +0.j]
 [ 0.5 -0.j   0.   +0.j -1.   +0.j   0.   +0.j]
 [ 0.25-0.j   0.   +0.j   0.   +0.j -0.9 +0.j]]
```

We have broken the energy level degeneracy with some terms added along the diagonal, and given a stronger coupling to the `(1,0)` state.

3.3.5 1.6 Decoherent transitions

There is one more important aspect of defining the `Sensor` graph that we have not discussed (which we can actually see as absent in the`Sensor` printout), and that is decoherent couplings. Decays of this sort are added with the `add_decoherence` function, which works very similarly to the `add_coupling` function. State manifolds and coupling coefficients work in an identical manner. Here we can copy the `Sensor` we used above, and add a decoherence in coupling the excited manifold to the ground state. The only minor difference is that decoherences do not use keyword arguments, but added with a single required argument defining the decay rate, again in Mrad/sec.

```
g = (0,0)
e = (1, [-1,0,1])
[e1, e2, e3] = rq.expand_statespec(e)

#defing coupling coefficients and energy shifts
cc = {
    (g,e1): 0.25,
    (g,e2): 0.5,
    (g,e3): 0.25
}
e_shifts = {e1:-0.1, e3: 0.1}

sensor_man = rq.Sensor([g, e])
sensor_man.add_energy_shifts(e_shifts)
sensor_man.add_coupling((g,e), detuning=1, rabi_frequency=2, coupling_
↪coefficients=cc, label="foo")
sensor_man.add_decoherence((e,g), 0.1, label="bar")
print(sensor_man.get_hamiltonian())
```

```
[[ 0.   +0.j   0.25+0.j   0.5 +0.j   0.25+0.j]
 [ 0.25-0.j -1.1 +0.j   0.   +0.j   0.   +0.j]
 [ 0.5 -0.j   0.   +0.j -1.   +0.j   0.   +0.j]
 [ 0.25-0.j   0.   +0.j   0.   +0.j -0.9 +0.j]]
```

Obviously, the hamiltonian will be the same, but we can inspect what got added to the graph again using `Sensor.couplings.edges`. We will then get the decoherence matrix with `decoherence_matrix`, similarly

to `get_hamiltonian()`. We will discuss how the decoherence matrix is used in more detail below when we discuss solving.

```
print("Graph Edges: ")
print(sensor_man.couplings.edges(data=True))
print("Decoherence matrix: ")
print(sensor_man.decoherence_matrix())
```

```
Graph Edges:
[((0, 0), (1, -1), {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0,␣
↪0), 'label': 'foo_0', 'coherent_cc': 0.25}), ((0, 0), (1, 0), {'rabi_frequency':␣
↪2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0, 0), 'label': 'foo_1', 'coherent_cc
↪': 0.5}), ((0, 0), (1, 1), {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec
↪': (0, 0, 0), 'label': 'foo_2', 'coherent_cc': 0.25}), ((1, -1), (1, -1), {'e_
↪shift': -0.1, 'label': '(1, -1)'}), ((1, -1), (0, 0), {'gamma_bar': 0.1, 'label
↪': '((1, -1),(0, 0))'}), ((1, 0), (0, 0), {'gamma_bar': 0.1, 'label': '((1, 0),
↪(0, 0))'}), ((1, 1), (1, 1), {'e_shift': 0.1, 'label': '(1, 1)'}), ((1, 1), (0,␣
↪0), {'gamma_bar': 0.1, 'label': '((1, 1),(0, 0))'})]
Decoherence matrix:
[[0.  0.  0.  0. ]
 [0.1 0.  0.  0. ]
 [0.1 0.  0.  0. ]
 [0.1 0.  0.  0. ]]
```

The edges are getting quite busy, but we will spot `"gamma_bar"` attributes on the expected graph edges. This is the convention used by `rydiqule`: when `add_decoherence` or a related function is called, and `label` is provided, the associated key on the graph will be `"gamma_" + label`. If no `label` is provided, simply `"gamma"` will be used for the key. When `decoherence_matrix` is called, every attribute on each edge that starts with `"gamma"` will be added together and placed on the appropriate term of a decoherence matrix.

There are a few more functions to be aware of when it comes to definining decoherence rates:

1. `add_transit_broadening` will add a decoherence from each state to the ground state (by default) or to a particular set of ground states. This function is designed to account for atoms which move out of a laser and are replaced by atoms in their ground state(s). Fractional transit broadenings are specified with the `repop` keyword arg.
2. `add_self_broadening` will add a decoherence from a state to itself to account for a atoms losing coherence; remaining in the same state.
3. `set_gamma_matrix` takes a square numpy array as an argument, then applies an appropriate `"gamma"` value to each graph edge.

We have printed attributes directly from the graph for the purposes of demonstrating what is happening under the hood. Unless you need to inspect attributes of the graph directly, a simple `print` statement will give us a more readable output.

```
print(sensor_man)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_0, coherent_cc: 0.25}
    ((0, 0),(1, 0)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_1, coherent_cc: 0.5}
    ((0, 0),(1, 1)): {rabi_frequency: 2, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_2, coherent_cc: 0.25}
Decoherent Couplings:
    ((1, -1),(0, 0)): {gamma_bar: 0.1}
```

(continues on next page)

(continued from previous page)

```
    ((1, 0),(0, 0)): {gamma_bar: 0.1}
    ((1, 1),(0, 0)): {gamma_bar: 0.1}
Energy Shifts:
    (1, -1): -0.1
    (1, 1): 0.1
```

3.4 2. Solving a `Sensor` in the steady-state case

So far we have just created sensor objects and shown their corresponding Hamiltonians. The most important functionality of `rydiqule`, however, is solving their associated equations of motion automatically. Here we will demonstrate some of the ways to solve a simple system using `rydiqule`.

3.4.1 2.1 What is `rydiqule` actually solving?

In the section title, we simply refer to "solving a sensor". When we discuss solving a system, what we mean is that we are solving the Lindblad Master Equation associated with the system described by our `Sensor`:

$$\dot{\rho} = -i[\hat{H}, \rho] - \mathcal{L}$$

where $\mathcal{L}$ is the Lindblad operator that accounts for non-unitary dephasing effects.

When `rydiqule` solves in the steady state, what is ultimately produced is the steady-state solution of that differential equation. `rydiqule` is capable of inferring the relevant matrix quantities from the graph defined by a `Sensor`, as we have seen above.

3.4.2 2.2 Basic Solving in the steady state

If all we need is steady state behavior of a particular system, it is straightforward and quick in `rydiqule`. By wrapping a simple matrix differential equation solver, `rydiqule` can quickly get the steady state frequencies of each element of the density matrix ρ of the system. We start by re-defining the same ladder system we had at the beginning of the notebook.

```
basis_size = 3
sensor_demo = rq.Sensor(basis_size)

laser_01 = {"states": (0,1), "detuning": 1, "rabi_frequency": 3}
laser_12 = {"states": (1,2), "detuning": 2, "rabi_frequency": 5}
sensor_demo.add_couplings(laser_01, laser_12)

gamma = np.zeros((basis_size, basis_size))
gamma[1:,0] = 0.1
print(gamma)
sensor_demo.set_gamma_matrix(gamma)
```

```
[[0.  0.  0. ]
 [0.1 0.  0. ]
 [0.1 0.  0. ]]
```

Now, instead of showing the Hamiltonian, we can just call the `rydiqule.solve_steady_state()` function on `sensor` and get the result.

```
solution_demo = rq.solve_steady_state(sensor_demo)
print(solution_demo)
```

```
<rydiqule.sensor_solution.Solution object at 0x000002333B42EE40>
```

3.4.3 2.3 The `Solution` object

As you can see, the solution itself is not an array, but an object containing some extra information. When `rydiqule` solves a system, it saves the solution, but also many other things relevant to the solve (including the graph itself), so that relevant quantities about the original `Sensor` and solve can be computed and reproduced. Since the above output is a bit cluttered, let's contain outselves to just the important quanties for now. First, `couplings` is just a copy of the `Sensor.couplings` graph discussed above for the `Sensor` on which `solve_steady_state` was called. Next, `rho` is the density matrix solution of the system, but there is a catch.

```
print(f"Couplings: {solution_demo.couplings}")
print(f"rho: {solution_demo.rho}")
print(len(solution_demo.rho))
```

```
Couplings: DiGraph with 3 nodes and 4 edges
rho: [-0.09225689 -0.1820376   0.01617957  0.27218216  0.21003648  0.00831697
  0.0042641   0.21320497]
8
```

Adjustments to the equations

So why does the solution have 8 elements? Why is it real? Why is it 1-dimensional? After all, it represents a density matrix, which should have n^2 complex-valued elements ($3^2 = 9$ in this case), ideally arranged in square.

For numerical stability, `rydiqule` removes the ground state population equations, so the ρ_{00} term is omitted from the solution (typically $\rho_{00} >> \rho_{ij}$ for $ij \neq 00$, which is the source of the numerical problems). If needed, it can be inferred as $\rho_{00} = 1 - \sum_{i=1}^{n} \rho_{ii}$. Furthermore, rydiqule transforms the basis so that all values are real. Rather than an $n \times n$ Hermitian matrix, `rydiqule` parameterizes the density matrix ρ as $n \times n$ real values before removing the ground state, for a total of $n^2 - 1$ values. For more details, refer to the documentation. We really do wish these modifications were not necessary, but keeping the ground state population and complex density matrix elememts resulted in unphysical soulutions during development.

So how do we know (ideally quickly) which element of the solution corresponds to which density matrix element? The `Solution` contains an attribute called `dm_basis` which does exactly this, containing a list of strings showing the info we want. Since the density matrix is hermitian and trace 1, this array does indeed contain all of the information of the density matrix. Note that `dm_basis` does not include ρ_{00} since the ground state is removed by the solver.

We choose to keep this basis in rydiqule since it is more transparent to what `rydiqule` does under the hood, and is used in further calculations.

```
print(solution_demo.dm_basis)
```

```
['10_real' '20_real' '10_imag' '11_real' '21_real' '20_imag' '21_imag'
 '22_real']
```

If we quickly want a particular full complex density matrix element, the `rho_ij` function does precisely that:

```
print(f"rho_10 = {solution_demo.rho_ij(1,0)}")
```

```
rho_10 = (-0.09225688789212251+0.016179570807727885j)
```

There are a few other functions to be aware of in a `Solution` object. Some commonly-computed quantities that might be derived from a density matrix solution. Before we discuss them, it is worth a brief aside on the `Solution.probe_tuple` quantity. It is a quantity that is derived from the `Sensor` that is the transition over which, by default, observable calculations will be made. So in the absence of any other information, when observable values are computed, they are computed for the density matrix elements that are defined by the `probe_tuple`. The `probe_tuple` is set by the first coupling that is added with the `add_coupling` function in `Sensor` (equivalently, the first dictionary

passed to `add_couplings`). While this quantity can be set manually, it is usually better practice to define whatever the probing transition (usually the laser you will be measuring) first, and not change `probe_tuple` manually when it can be avoided. To demonstrate this, we can see what `probe_tuple` is in both the `Sensor` and `Solution`.

```
print(sensor_demo.probe_tuple)
print(solution_demo.probe_tuple)
```

```
(0, 1)
(0, 1)
```

As expected, this is the the first coupling added to the system. Below is a (very) brief example to show the `probe_tuple` being added.

```
sensor_probe = rq.Sensor(3)
print(sensor_probe.probe_tuple)
sensor_probe.add_coupling((0,2), rabi_frequency=1, detuning=0)
print(sensor_probe.probe_tuple)
sensor_probe.add_coupling((1,2), rabi_frequency=1, detuning=0)
print(sensor_probe.probe_tuple)
```

```
None
(0, 2)
(0, 2)
```

With the `probe_tuple` cleared up, we can explore a few of the other functions in a `Solution`.

1. `get_observable(A)` is a function which, given an observable matrix A, uses the density matrix definition of the expectation value of A, $\langle A \rangle = tr(\rho A)$.
2. `coupling_coefficent_matrix(<coupling>)` returns a matrix of all coupling coefficients on a particular transition defined by coupling. We discussed the coupling cofficients above in the context of relative coupling strength. In this way they can be thought of as a unitless analogue of the dipole moment.
3. Given these two quantities, you may be developing an idea that it my be useful to pass the output of `coupling_coefficient_matrix` to `get_observable` and use that to compute susceptibility. It's such a good idea, `rydiqule` went ahead and implemented that function in `Solution.get_susceptibility()`. The value will be computed from a handful of quantities built in to `Solution`, and more details about its calculation can be found in the docs. Note that this calculation will require setting the `kappa` and `probe_freq` attributes in a `Sensor` before it is solved (or, technically, in the solution itself but this is not the recommended use). Again, see the docs for further details. Note that this function will implicitly only get the observable over the `probe_tuple` transition.

We will demonstrate getting the susceptibility with `get_susceptibility()` below, although keep in mind these are *very* cartoonish and unphysical values

```
sensor_demo.set_experiment_values(kappa=1, probe_freq=1000)
solution_demo = rq.solve_steady_state(sensor_demo)
print(solution_demo.get_susceptibility())
```

```
(-0.03687722558481313+0.006467351069111717j)
```

3.4.4 2.4 Stacking: Solving for multiple values of a parameter

Getting a single result is nice, but really just a single data point. One of the simplest types of experiments you might want to simulate is to scan, for example, over a range of detuning values and see the behavior of the system at each value. One of `rydiqule`'s greatest strength is its handling of simulations like this. Suppose we wanted to sweep our probe laser detuning between -10 MHz and 10 MHz. Without `rydiqule` we might write an explicit loop over a set of values, modifying and solving the system at each iteration, and storing the results in a new list. In python especially, this sort of approach has a high computational overhead.

With rydiqule, we can set up the system similarly to above, this time as a 4-level system. However, this time, instead of using a single float value for a coupling parameter, we will define the $0 \leftrightarrow 1$ coupling detunuing value as a 1-dimensional numpy array constructed with the linspace function. after this, rydiqule will handle the rest.

```
basis_size = 4
sensor_sweep = rq.Sensor(basis_size)

detunings = 2*np.pi*np.linspace(-10,10,201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7}

sensor_sweep.add_couplings(probe, coupling, rf)

sensor_sweep.add_decoherence((1,0), 0.1)
sensor_sweep.add_decoherence((2,1), 0.1)
sensor_sweep.add_decoherence((3,0), 0.1)
```

Before we continue further, let us examine the printout of the Sensor, and the shape of the hamiltonian produced by get_hamiltonian:

```
print(sensor_sweep)
print(sensor_sweep.get_hamiltonian().shape)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 3, detuning: <parameter with 201 values>, phase: 0,␣
↪kvec: (0, 0, 0), coherent_cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 5, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 7, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (2,3)}
Decoherent Couplings:
    (1,0): {gamma: 0.1}
    (2,1): {gamma: 0.1}
    (3,0): {gamma: 0.1}
Energy Shifts:
    None
(201, 4, 4)
```

Here we see the familiar 4×4 hamiltonian, but with an additional dimension of size 201, exaclty the size of our detuning array. What rydiqule has done behind the scenes is generate a 4×4 Hamiltonian matrix for every detuning value in the array and place them all into a single np.ndarray, enabling the use of fast broadcasted numpy functions. rydiqule calls this "stacking", and it is the key behind rydiqule's impressive python performace for complicated experiments. Most (although not all) parameters of a Sensor can be scanned in this way. The complete list of scannable parameters is:

1. detuning
2. rabi_frequency
3. phase
4. e_shift
5. All decoherences

An important note about rydiqule's internal functions is that they are completely agnostic to what each axis of a stack represents, when run through the internal functions, all it needs to know is that something is a stack of

hamiltonians. Equations, or density matrices, and the fuctions that use them are performed identically. This will be an important detail in later sections.

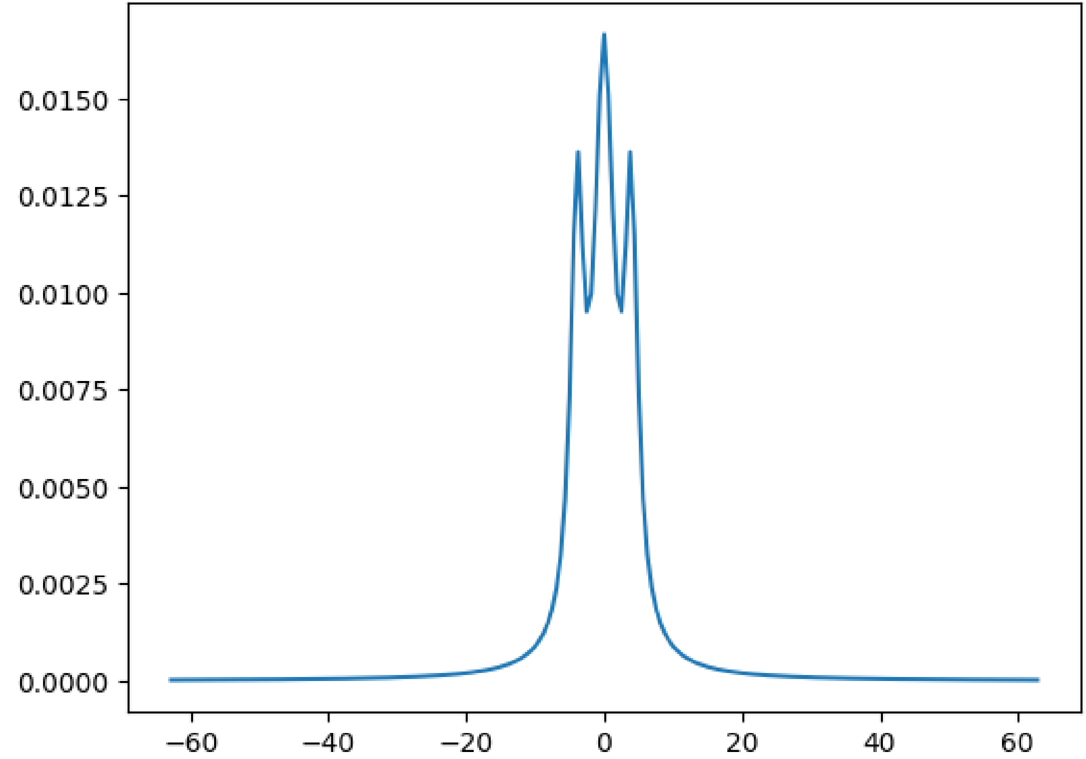
From here, we solve exactly as before. Rydiqule will automagically solve the system for every detuning value (very quickly). We also show the utility function `get_rho_ij()` which extracts ρ_{ij} from the density matrix ρ of any solution (regardless of how many dimensions it is). We use it to get ρ_{01}.

```
solution = rq.solve_steady_state(sensor_sweep)
print(f"Solution shape: {solution.rho.shape}")
absorption = solution.coupling_coefficient_observable().imag
print(f"Absorption_shape: {absorption.shape}")
```

```
Solution shape: (201, 15)
Absorption_shape: (201,)
```

We can see that there is a 15-element solution for each one of the detuning values. After we get the absorption using `get_rho_ij()`, we can see that we are left with a single array with 201 elements, corresponding to to the absorption at each value of detuning. Still, that was pretty fast for solving 200 equations. The function doesn't do anything special, it just is a quick way to get common info you might want out of a solution. We can now do a quick-and-dirty plot to see what it looks like:

```
plt.plot(detunings, absorption)
plt.show()
```



3.4.5 2.5 Solving for multiple values of multiple parameters

If you like the simplicity of scanning an single laser detuning in `rydiqule`, you will be even more excited to learn that rydiqule can handle scans over multple different parameters simulataneously. We will set up the sensor as before, with a couple of changes. We will now also scan the dephasing rate and show how the absorption changes.

```
basis_size = 4
sensor_sweep_2 = rq.Sensor(basis_size)
```

(continues on next page)

(continued from previous page)

```
detunings = 2*np.pi*np.linspace(-10, 10, 201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3, "label":"probe
↪"}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5, "label": "coupling
↪"}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7, "label": "rf"}

sensor_sweep_2.add_couplings(probe, coupling, rf)

gamma10 = np.linspace(0.1, 1.0, 10)
sensor_sweep_2.add_decoherence((1,0), gamma10)
sensor_sweep_2.add_decoherence((2,1), 0.1)
sensor_sweep_2.add_decoherence((3,0), 0.1)

print(sensor_sweep_2)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 3, detuning: <parameter with 201 values>, phase: 0,␣
↪kvec: (0, 0, 0), label: probe, coherent_cc: 1.0}
    (1,2): {rabi_frequency: 5, detuning: 0, phase: 0, kvec: (0, 0, 0), label:␣
↪coupling, coherent_cc: 1.0}
    (2,3): {rabi_frequency: 7, detuning: 0, phase: 0, kvec: (0, 0, 0), label: rf,␣
↪coherent_cc: 1.0}
Decoherent Couplings:
    (1,0): {gamma: <parameter with 10 values>}
    (2,1): {gamma: 0.1}
    (3,0): {gamma: 0.1}
Energy Shifts:
    None
```

Once again, no extra steps are required, just call `solve_steady_state()` as normal, and look at the shape of the solution and the absorption and rydiqule will quickly find the solution for every combination of those 2 parameters.

```
solution_2 = rq.solve_steady_state(sensor_sweep_2)
print(f"Solution shape: {solution_2.rho.shape}")
absorption_2 = solution_2.coupling_coefficient_observable().imag
print(f"Absorption_shape: {absorption_2.shape}")
```

```
Solution shape: (201, 10, 15)
Absorption_shape: (201, 10)
```

How do we know which axis corresponds to the probe detuning and which is the coupling detuning? It is obvious in this case since they are different lengths, but this will not always be the case (for example in cases where two lasers are scanned over the same number of values). `Sensor` and `Solution` both have attributes called `axis_labels` which do just that, and this is where labeling our couplings comes in handy (It is a method of `Sensor`and an attribute of `Solution` for reasons that are not important here). Note that if couplings are not labeled, the axes will default to being labeled by the states they couple. For example `["(0,1)_detuning", (1,2)_detuning"]`, in this case, the output of `axis_labels` is what we would expect based on the solution shape we printed above. Notice that since the detuning scanned was on the laser we labelled `"probe"`, that label is reflected on the axis.

In the case of a `Solution`, the axis of the density matrix is included, but this is not the case in `Sensor`, since there is no associated density matrix until it is solved.

```
print(sensor_sweep_2.axis_labels())
print(solution_2.axis_labels)
```

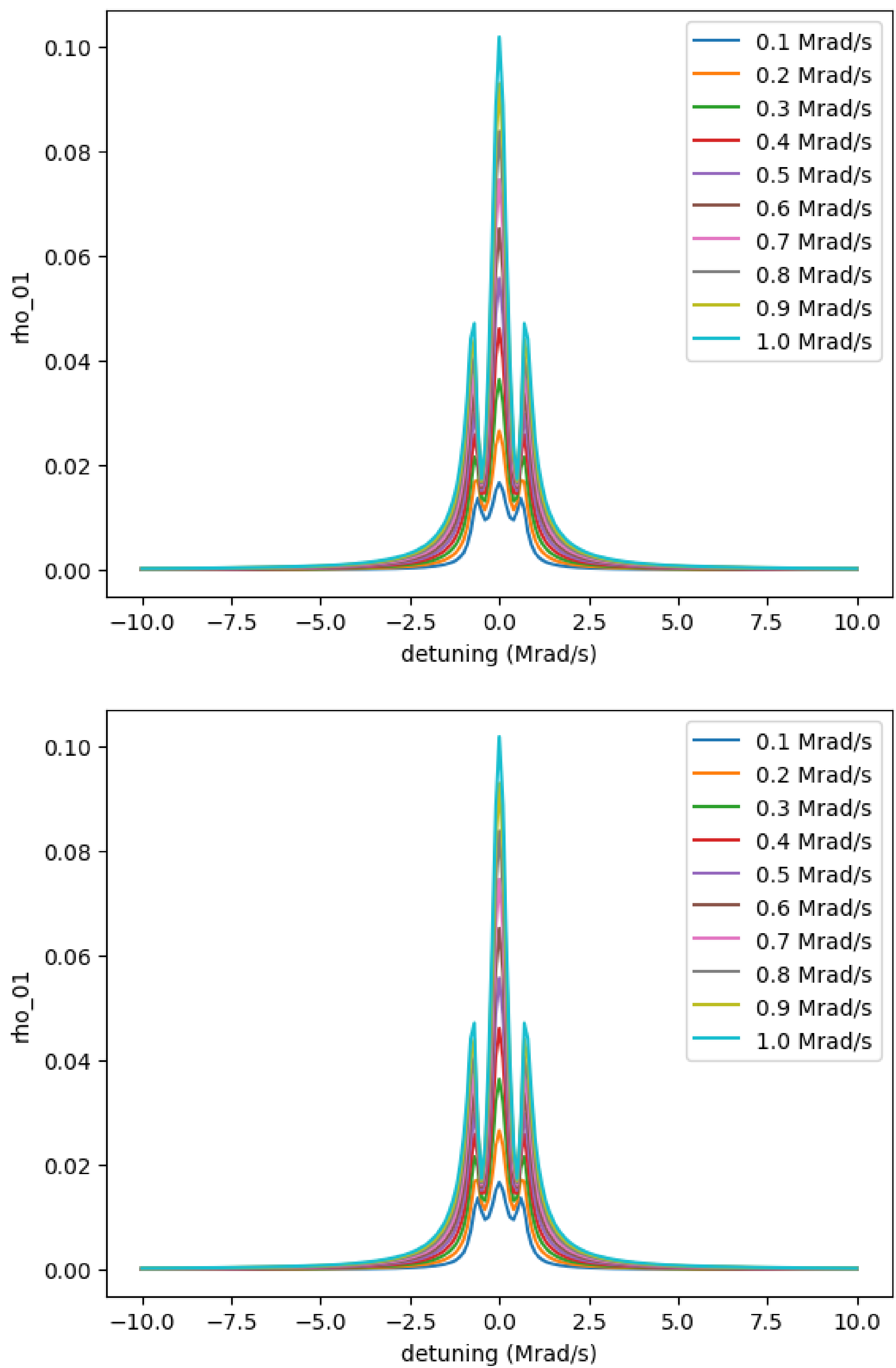
```
['probe_detuning', '(1,0)_gamma']
['probe_detuning', '(1,0)_gamma', 'density_matrix']
```

So the first axis corresponds to the probe laser detuning, and the second axis corresponds to $\Gamma_{1,0}$. Now calling a function like `np.argmin()` or `np.argmax()` could quickly be used to optimize some value over a large parameter space. `rydiqule` lets us add as many parameters as you like as a list, detunings, rabi frequencies, or dephasings. It's handy, but beware that the memory footprint can quickly balloon out of control when generating equations of motion if you get too ambitious, increasing by a factor of n when you add an axis with n elements, especially given that many values used in internal calculations are 128-bit complex arrays.

Currently, `rydiqule` has internal functions to spot this before it happens and break it into more manageable chunks if it can, but it is certainly possible to make a system which even it cannot handle. If the solution does not fit in memory, no amount of splitting up the equations will allow the system to be solved, and you should reconsider the size of your parameter space. Even in the above example, `rydiqule` is solving $201 \times 10 \geq 2,000$ equations simultaneously (which is still pretty good). In cases where the parameter space would become intractably large, it is generally good practice to optimise over specific subsets of parameters at a time, or use a coarser mesh to identify a smaller region of interest.

We can plot these probe sweeps at the same time fairly simply.

```
fig, ax = plt.subplots(1)

for i in range(absorption_2.shape[1]):
    ax.plot(detunings/2/np.pi, absorption_2[:,i], label=f'{gamma10[i]:.1f} Mrad/s')

ax.set_xlabel("detuning (Mrad/s)")
ax.set_ylabel("rho_01")

ax.legend();
fig
```


0.1 Mrad/s
0.2 Mrad/s
0.3 Mrad/s
0.4 Mrad/s
0.5 Mrad/s
0.6 Mrad/s
0.7 Mrad/s
0.8 Mrad/s
0.9 Mrad/s
1.0 Mrad/s
rho_01
detuning (Mrad/s)
0.1 Mrad/s
0.2 Mrad/s
0.3 Mrad/s
0.4 Mrad/s
0.5 Mrad/s
0.6 Mrad/s
0.7 Mrad/s
0.8 Mrad/s
0.9 Mrad/s
1.0 Mrad/s
rho_01
detuning (Mrad/s)

3.4.6 2.6 Parameter zipping

It is not uncommon in atomic physics experiments for 2 different atomic transitions (even just among sublevels) to originate from the same laser. In this case, it is often desirable to run simulations where a value of a parameter in one transition (say the laser power/rabi frequency) maps 1-to-1 to a value in a different transition. At the very least, ensuring that these values are always run together will reduce the number of calculations that need to be done substantially by elimnating unphysical simulations.

Zipping Basics

To handle this case, `Sensor` has a function called `zip_parameters`. We call it a zip because it can be thought about in the same way that python's native `zip()` works for iterators. A zip is defined with a dictionary keyed with coupling labels (including the automatically-generated ones like `"(1,2)"`). `zip_parameters` takes this dictionary as an argument, then places all those values on the same axis in the hamiltonian and equation stack. We demonstrate it below:

```
sensor_zip = rq.Sensor(3)
det = np.linspace(-5,5,101)
f1 = {"states": (0,1), "rabi_frequency": 1, "detuning": det, "label":"laser"}
f2 = {"states": (1,2), "rabi_frequency": 1, "detuning": 2*det}
sensor_zip.add_couplings(f1, f2)
print(f"shape before zip: {sensor_zip.get_hamiltonian().shape}")
sensor_zip.zip_parameters({"laser":"detuning", "(1,2)":"detuning"}, zip_label="foo
↪")
print(f"shape after zip: {sensor_zip.get_hamiltonian().shape}")
```

```
shape before zip: (101, 101, 3, 3)
shape after zip: (101, 3, 3)
```

We labelled the zip as `"foo"`, and this is what will be shown on the output of `"axis_labels"`. Had we not provided a label, the one created would be all the couplings joined with underscores. This can be cumbersome-looking, so it generally good practice to label your zips, although not strictly mandatory.

```
sensor_zip.add_transit_broadening(0.1)
sol = rq.solve_steady_state(sensor_zip)
print(sol.axis_labels)
```

```
['foo', 'density_matrix']
```

Now all the parameters we zipped should line up on their shared axis. As an example, if we inspect Hamiltonian 0, it should correspond to a detuning of -5 (the first detuning value) on the first coupling and -10 (the first detuning value) on the second:

```
print(sensor_zip.get_hamiltonian()[0])
```

```
[[ 0. +0.j  0.5+0.j  0. +0.j]
 [ 0.5-0.j  5. +0.j  0.5+0.j]
 [ 0. +0.j  0.5-0.j 15. +0.j]]
```

It is worth noting that `zip_parameters` does not change anything on the graph itsself, it just adds the labels to a list which is an internal attribute. The actual zipping is done at hamiltonian/decoherence matrix generation time.

Automatic Zipping

When couplings are defined between manifolds of states as shown above (thus producing more than 1 graph edge) any array-like parameters will be zipped automatically. So no additional steps need to be taken, and the shape of the Hamiltonian will be correct. Here, we see that all the detunings are included as arrays on separate edges, but they share an axis on the hamiltonian automatically.

```
g = (0,0)
e = (1, [-1,0,1])
[e1, e2, e3] = rq.expand_statespec(e)

#defing coupling coefficients and energy shifts
cc = {
    (g,e1): 0.25,
    (g,e2): 0.5,
    (g,e3): 0.25
}
e_shifts = {e1:-0.1, e3: 0.1}

det = np.linspace(-1,1,11) #short array for readable printout

sensor_man_zip = rq.Sensor([g,e])
sensor_man_zip.add_energy_shifts(e_shifts)
sensor_man_zip.add_coupling((g,e), rabi_frequency=2, detuning=det, coupling_
↪coefficients=cc, label="laser")

print(sensor_man_zip)

print(f"Hamiltonian Shape :{sensor_man_zip.get_hamiltonian().shape}")
print(sensor_man_zip.axis_labels())
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser_0, coherent_cc: 0.25, laser_detuning:␣
↪detuning}
    ((0, 0),(1, 0)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser_1, coherent_cc: 0.5, laser_detuning:␣
↪detuning}
    ((0, 0),(1, 1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser_2, coherent_cc: 0.25, laser_detuning:␣
↪detuning}
Decoherent Couplings:
    None
Energy Shifts:
    (1, -1): -0.1
    (1, 1): 0.1
Zip Labels:
    ['laser_detuning']
Hamiltonian Shape :(11, 4, 4)
['laser_detuning']
```

It is also worth mentioning that while it handles a more sepcializaed use-case, `rydiqule` does not place any restrictions on what types of parameters can be zipped together. So as long as their lengths match, the `rabi_frequency` of one coupling can be zipped to `e_shift` on a completely different node for example.

Zipping zips

You can also zip a zip to another zip. While this sentence sounds a little ridiculous, it can be useful in cases where, for example, two coupling groups representing two different polarizations originate from the same laser. If the couplings to be zipped are over manifolds, they are already zipped amongst themselves, since zips are created automatically. But we still want to zip further. The exact use cases for this will be left to a more advanced example in a different notebook, but we will demonstrate the basic idea and functionality below. We start by creating two coupling groups, each with a scanned detuning.

```
g = (0,0)
e1 = (1,[-1,1])
e2 = (2,[-1,1])

det = np.linspace(-1,1,11)

sensor_zip_zip = rq.Sensor([g, e1, e2])
sensor_zip_zip.add_coupling((g, e1), detuning=det, rabi_frequency=1, label="laser1
↪")
sensor_zip_zip.add_coupling((g, e2), detuning=det, rabi_frequency=2, label="laser2
↪")
print(sensor_zip_zip)
print(f"Hamiltonian Shape: {sensor_zip_zip.get_hamiltonian().shape}")
print(f"Axis Labels: {sensor_zip_zip.axis_labels()}")
```

```
<class 'rydiqule.sensor.Sensor'> object with 5 states and 4 coherent couplings.
States: [(0, 0), (1, -1), (1, 1), (2, -1), (2, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 1, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser1_0, coherent_cc: 1.0, laser1_detuning:␣
↪detuning}
    ((0, 0),(1, 1)): {rabi_frequency: 1, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser1_1, coherent_cc: 1.0, laser1_detuning:␣
↪detuning}
    ((0, 0),(2, -1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser2_0, coherent_cc: 1.0, laser2_detuning:␣
↪detuning}
    ((0, 0),(2, 1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser2_1, coherent_cc: 1.0, laser2_detuning:␣
↪detuning}
Decoherent Couplings:
    None
Energy Shifts:
    None
Zip Labels:
    ['laser1_detuning', 'laser2_detuning']
Hamiltonian Shape: (11, 11, 5, 5)
Axis Labels: ['laser1_detuning', 'laser2_detuning']
```

We can see that we have already created 2 distinct zips implicitly in `add_coupling`, but we would like them to be zipped together further. We can use the `zip_zips` function to zip them together. It works as you might expect, taking as arguments the labels for any zips you would like to zip together further. Optionally, you can (and probably should) also add a new label.

```
sensor_zip_zip.zip_zips("laser1_detuning", "laser2_detuning", new_label="laser_
↪detuning")
print(sensor_zip_zip)
print(f"Hamiltonian Shape: {sensor_zip_zip.get_hamiltonian().shape}")
print(f"Axis Labels: {sensor_zip_zip.axis_labels()}")
```

```
<class 'rydiqule.sensor.Sensor'> object with 5 states and 4 coherent couplings.
States: [(0, 0), (1, -1), (1, 1), (2, -1), (2, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 1, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser1_0, coherent_cc: 1.0, laser_detuning:␣
↪detuning}
```

(continues on next page)

(continued from previous page)

```
    ((0, 0),(1, 1)): {rabi_frequency: 1, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser1_1, coherent_cc: 1.0, laser_detuning:␣
↪detuning}
    ((0, 0),(2, -1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser2_0, coherent_cc: 1.0, laser_detuning:␣
↪detuning}
    ((0, 0),(2, 1)): {rabi_frequency: 2, detuning: <parameter with 11 values>,␣
↪phase: 0, kvec: (0, 0, 0), label: laser2_1, coherent_cc: 1.0, laser_detuning:␣
↪detuning}
Decoherent Couplings:
    None
Energy Shifts:
    None
Zip Labels:
    ['laser_detuning']
Hamiltonian Shape: (11, 5, 5)
Axis Labels: ['laser_detuning']
```

This should cover all the important functionality regarding handling arrays of parameters in `Sensor`. This is not an exhaustive description of the functionality, for that please consult the docs. This discussion should, however, give you an excellent foundation for building your own experiments in `rydiqule` and give you some familiarity with the language used throughout the documentation.

3.5 3. Solving in the time domain

Depending on your experiment, you may be interested in the behavior of an atom acting under the influence of a field that varies with time.

Suppose you have defined a sensor, and we want to see the response of the sensor of some well-defined rf input function. In this case, the steady-state solution above is not enough. We will set up a sensor similar to the one above, but define the rf field differently. We will use the `"time_dependence"` keyword in the dictionary, and supply a regular python function which takes a single argument of time in microseconds. The output of this function should be a scalar quantity which will applied as a multiplicative factor to `rabi_frequency` at the time supplied. See `rydiqule`'s physics documentation for more details about the math of how time dependence is handled.

3.5.1 3.1 Time Dependence without the RWA

We will begin our discussion of the time solver by solving a problem where we do not make use of the rotating wave approximation (RWA) for the time-dependant field, since this is a more straightforward case. All we need to do to tell `rydiqule` not to use the rotating wave approximation is not supply a detuning value. Rather we define a transition frequency between two states and supply a python function of a single variable (which `rydiqule` interprets as time `t` in μs), and returns the complex scalar value by which to multiply the `rabi_frequency` at time `t`. Unlike in the RWA case, no implicit assumptions about the coupling will be made; it is an "empty" transition without a `time_dependence` function.

In the simple example below, we (arbitrarily) define a the 100 Mrad/s transtion, and we define a function `my_field_norwa` which applies a sinusoidal field with 1 Mrad/s detuning, modulated by a 10 Mrad/s carrier.

```
#101 Mrad/s field, modulated
def my_field_norwa(t):
    return np.cos(101*t)*np.sin(10*t)

sensor_time_norwa = rq.Sensor(4)
sensor_time_norwa.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_norwa.add_coupling((1,2), detuning=0, rabi_frequency=2)
```

(continues on next page)

(continued from previous page)

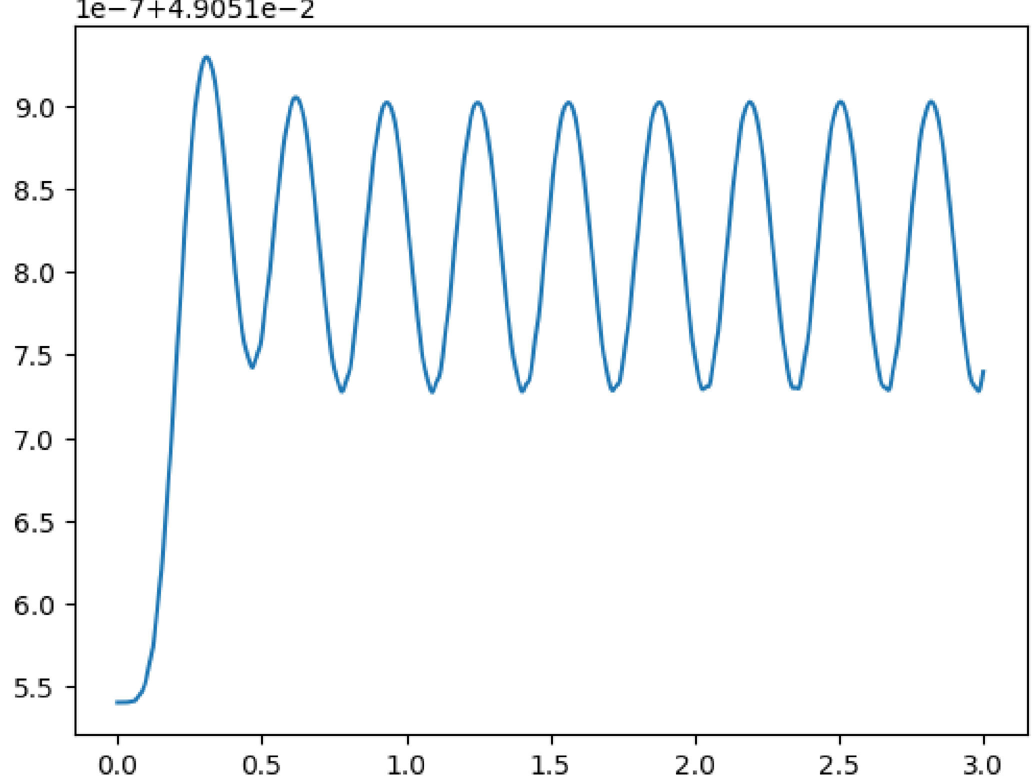
```
sensor_time_norwa.add_coupling((2,3), transition_frequency=100, rabi_frequency=1,␣
↪time_dependence=my_field_norwa)

sensor_time_norwa.add_transit_broadening(10)
sensor_time_norwa.add_decoherence((2,1), 1)
sensor_time_norwa.add_decoherence((3,2), 0.1)

print(sensor_time_norwa)

#1000 samples over 3 microsecond
end_time = 3
num_pts = 1000

sol_norwa = rq.solve_time(sensor_time_norwa, end_time, num_pts)
print(type(sol_norwa))
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 2, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 1, transition_frequency: 100, kvec: (0, 0, 0), time_
↪dependence: <function my_field_norwa at 0x000002333D6727A0>, coherent_cc: 1.0,␣
↪label: (2,3)}
Decoherent Couplings:
    (0,0): {gamma_transit: 10.0}
    (1,0): {gamma_transit: 10.0}
    (2,0): {gamma_transit: 10.0}
    (2,1): {gamma: 1.0}
    (3,0): {gamma_transit: 10.0}
    (3,2): {gamma: 0.1}
Energy Shifts:
    None
<class 'rydiqule.sensor_solution.Solution'>
```

This is a very straightforward example with simple numbers to demonstrate using the functionality of the time solver. The system was not set up in a precise enough way to warrant any serious interpretation of the output (although it does visually settle into steady-state behavior), but we can see that it produces an object of type `Solution`, just like a steady-state solve. Although it is the same type of object, it has some new attributes, some of which are demonstrated below.

```
print(f"rho shape: {sol_norwa.rho.shape}")
print(sol_norwa.axis_labels)
print(len(sol_norwa.t))

susc_norwa=sol_norwa.coupling_coefficient_observable()
plt.plot(sol_norwa.t, susc_norwa.imag)
plt.show()
```

```
rho shape: (1000, 15)
['time', 'density_matrix']
1000
```



3.5.2 3.2 Initial conditions

Obviously, it does not make sense to discuss a time-domain solve without discussing the initial condition of the system. Since no explicit condition was defined in the above example, it begs the questions of 1. What intial condition was used? and 2. Is there any way to use a different one? Thankfully, there are simple answers to both questions. Let's recreate the example 3.1.

```
#101 Mrad/s field, modulated
def my_field_norwa(t):
    return np.cos(101*t)*np.sin(10*t)

sensor_time_norwa = rq.Sensor(4)
sensor_time_norwa.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_norwa.add_coupling((1,2), detuning=0, rabi_frequency=2)
sensor_time_norwa.add_coupling((2,3), transition_frequency=100, rabi_frequency=1,␣
↪time_dependence=my_field_norwa)

sensor_time_norwa.add_transit_broadening(10)
sensor_time_norwa.add_decoherence((2,1), 1)
sensor_time_norwa.add_decoherence((3,2), 0.1)

#1000 samples over 3 microsecond
end_time = 3
num_pts = 1000

sol_norwa = rq.solve_time(sensor_time_norwa, end_time, num_pts)
```

Thankfully, the initial condition used is actually saved in the `Solution` object when `solve_time` is called under the `init_cond` attribute:

```
print(sol_norwa.init_cond)
```

```
[-0.         -0.00463991  0.          0.04905154  0.00478423 -0.
  0.         -0.          0.00066507  0.00012092  0.         -0.
 -0.         -0.         -0.        ]
```

This density matrix (remember that the basis of the density matrix is altered in `rydiqule`) is the steady-state solution of the system based on the Hamiltonian at time `t=0`, as we can see below. We will introduce the `get_time_hamiltonian` function, which is not crucial to know, but will demostrate where the result comes from.

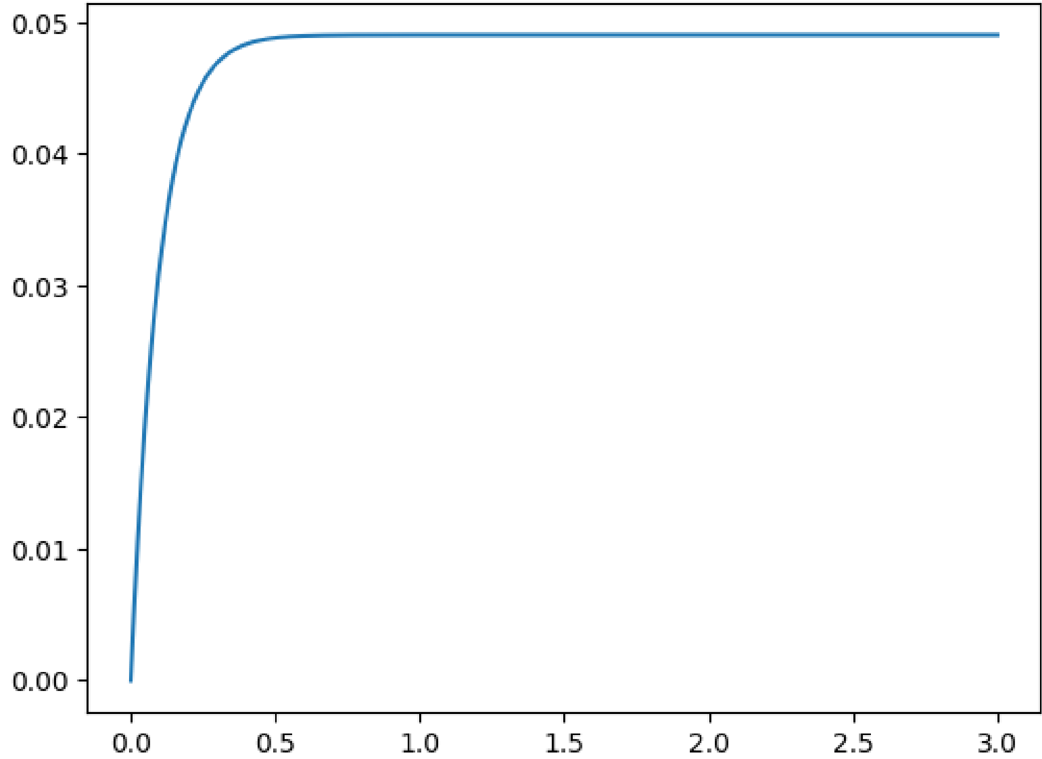
```
print(sensor_time_norwa.get_time_hamiltonian(0))
print(rq.solve_steady_state(sensor_time_norwa).rho)
```

```
[[  0. +0.j   0.5+0.j   0. +0.j   0. +0.j]
 [  0.5+0.j   0. +0.j   1. +0.j   0. +0.j]
 [  0. +0.j   1. +0.j   0. +0.j   0. +0.j]
 [  0. +0.j   0. +0.j   0. +0.j 100. +0.j]]
[-0.         -0.00463991  0.          0.04905154  0.00478423 -0.
  0.         -0.          0.00066507  0.00012092  0.         -0.
 -0.         -0.         -0.        ]
```

So the steady-state solution (which uses the `t=0` hamiltonian for a steady-state sole when there is a time-dependant coupling) is the same as the the initial condition.

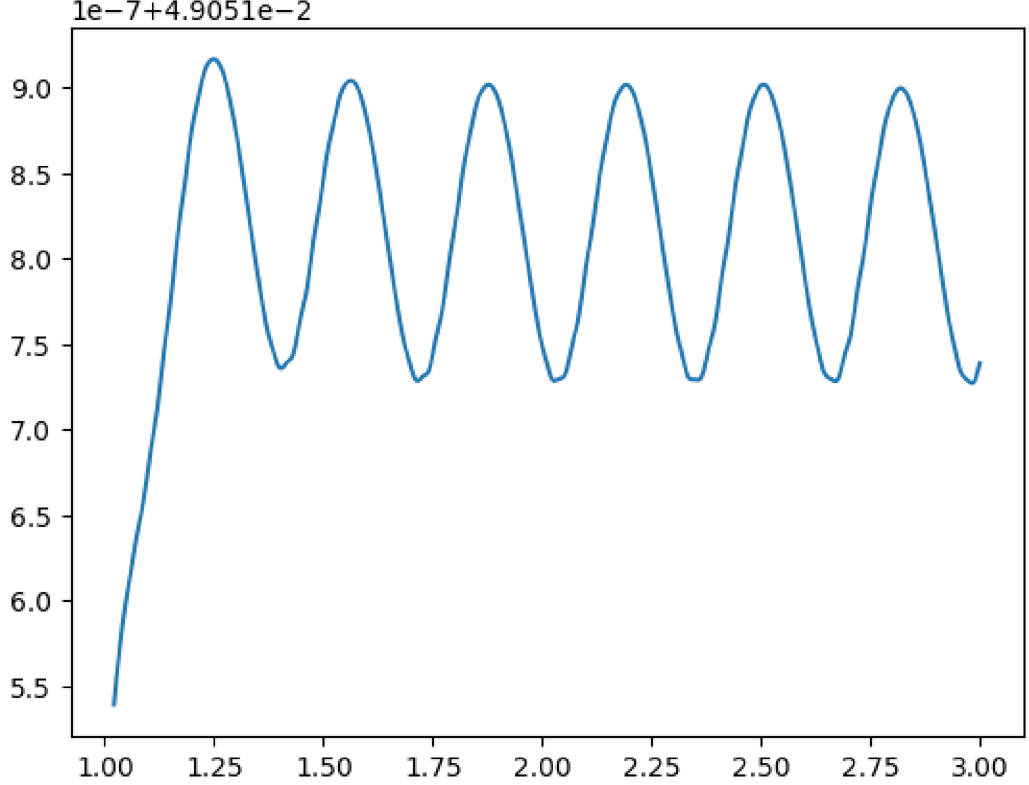
Setting the intial condition manually is as simple as using the `init_cond` argument in `rq.solve_time`. In this case, we will demonstrate a simple case where all population is in the ground states. Remember that since `rydiqule` removes the ground state population when it solves, this corresponds to an array of zeros with length $n^2 - 1 = 15$. If you would like a more involved intial condition, you can create the familiar square complex (hermitian) density matrix, and call `rq.sensor_utils.convert_complex_to_dm()` to get the corresponding real flattened density matrix.

```
all_ground = np.zeros(4**2-1)
sol_norwa_ground = rq.solve_time(sensor_time_norwa, end_time, num_pts, init_
↪cond=all_ground)
susc_norwa_ground = sol_norwa_ground.coupling_coefficient_observable()

fig, ax = plt.subplots()
ax.plot(sol_norwa_ground.t, susc_norwa_ground.imag)
plt.show()
```



This plot helps shed some light on the reason `rydiqule` makes the default choice it does. The transient ramp up is much bigger, and it is hard to see any oscillations in this plot. Depending on what you are modelling, it often makes sense to leave the inital condition alone, but it is useful to know you can change it. Let's chop off the first 340 points and see that the steady-state behavior is still similar:

```
fig, ax = plt.subplots()
ax.plot(sol_norwa_ground.t[340:], susc_norwa_ground.imag[340:]);
plt.show()
```



3.5.3 3.3 Time Dependance with the RWA

Now that we have shown time-dependence in the simplest case, with no rotating wave approximation, let us examine how we can still use the rotating wave approximation and still solve for time-domain behavior of the system. This is desirable in a number of cases, for example in a pulsed laser scheme.

Remember that if the RWA is valid for the system you are trying to model, it is generally a good idea to use it. `rydiqule` does note make any decisions regarding this for you, it is up to the physicist to decide in which situations it is a reasonable approximation of system behavior. In any time solver, modelling higher frequencies will be slower. Depending on how high your frequencies are, this can be many orders of magnitude. For this reason, modelling lasers fully in the time domain, for example is basically always a bad idea; you might be waiting an amount of time that is, practically speaking, forever.

Below we show a system that we define similarly to the previous example, the only exception being that the `(2,3)` coupling now is specified with a 1 Mrad/s detuning (the same detuning defined explicitly above), and show that the behavior is the same.

```
#101 Mrad/s field, modulated
def my_field_rwa(t):
    return np.sin(10*t)

sensor_time_rwa = rq.Sensor(4)
sensor_time_rwa.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_rwa.add_coupling((1,2), detuning=0, rabi_frequency=2)
sensor_time_rwa.add_coupling((2,3), detuning=1, rabi_frequency=1, time_
↪dependence=my_field_rwa)

sensor_time_rwa.add_transit_broadening(10)
sensor_time_rwa.add_decoherence((2,1), 1)
sensor_time_rwa.add_decoherence((3,2), 1)

print(sensor_time_rwa)
```

(continues on next page)

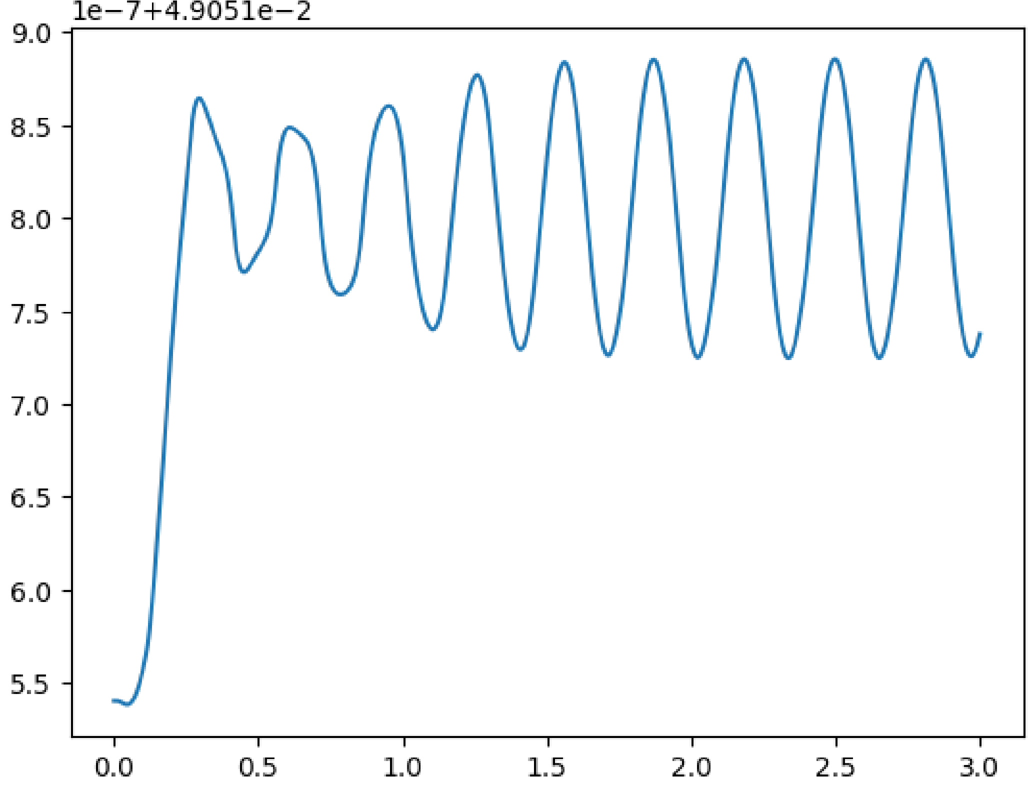
(continued from previous page)

```
#1000 samples over 3 microsecond
end_time = 3
num_pts = 1000

sol_rwa = rq.solve_time(sensor_time_rwa, end_time, num_pts)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 2, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0), time_
↪dependence: <function my_field_rwa at 0x000002333D6FC9A0>, coherent_cc: 1.0,␣
↪label: (2,3)}
Decoherent Couplings:
    (0,0): {gamma_transit: 10.0}
    (1,0): {gamma_transit: 10.0}
    (2,0): {gamma_transit: 10.0}
    (2,1): {gamma: 1.0}
    (3,0): {gamma_transit: 10.0}
    (3,2): {gamma: 1.0}
Energy Shifts:
    None
```

Just like before, we will print the imaginary part of a unitless “susceptibility” against time, and see that indeed we get similar transient and quick steady-state behavior.

```
fig, ax = plt.subplots()
susc=sol_rwa.coupling_coefficient_observable()
ax.plot(sol_rwa.t, susc.imag);
plt.show()
```



There is an important caveat with regards to how to think about RWA couplings in the time domain. Remember that the approximation treats the field in the rotating frame of the *field*. In `rydiqule` this applied field is implicitly assumed to be part of the coupling. This is how we keep specifications of couplings quick and easy. So in the above example, our 101 Mrad/s field is rotated into the 101 Mrad/s frame, with of 1 Mrad/s detuning from resonance (accounted for on the diagonal). As a result, the $\cos(101t)$ field becomes 1 as far as rydiqule is concerned, and we are left only with the modulating sin function and the manually specified detuning of 1 Mrad/s.

One should also note that there are some obvious numeric artifacts. Their origin and how to mitigate them is described in section 3.4.

Any python function works

Since `rydiqule` just calls the provided time function with no inference of its mathematical form, any python funtion will work; it need not be a sinusoid. This opens up options for things like stepwise functions. Lets reproduce the above RWA example, but rather than modulating the field by a sinusoid, we will use a pulsing function.

```
# off from 0 to 1, on from 1 to 2, repeat
def pulse_field(t):
    if 0 <= t%2 <= 1:
        return 0.0
    else:
        return 1.0

sensor_time_pulse = rq.Sensor(4)
sensor_time_pulse.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_pulse.add_coupling((1,2), detuning=0, rabi_frequency=2)
sensor_time_pulse.add_coupling((2,3), detuning=1, rabi_frequency=1, time_
↪dependence=pulse_field)

sensor_time_pulse.add_transit_broadening(10)
sensor_time_pulse.add_decoherence((2,1), 1)
sensor_time_pulse.add_decoherence((3,2), 1)
```

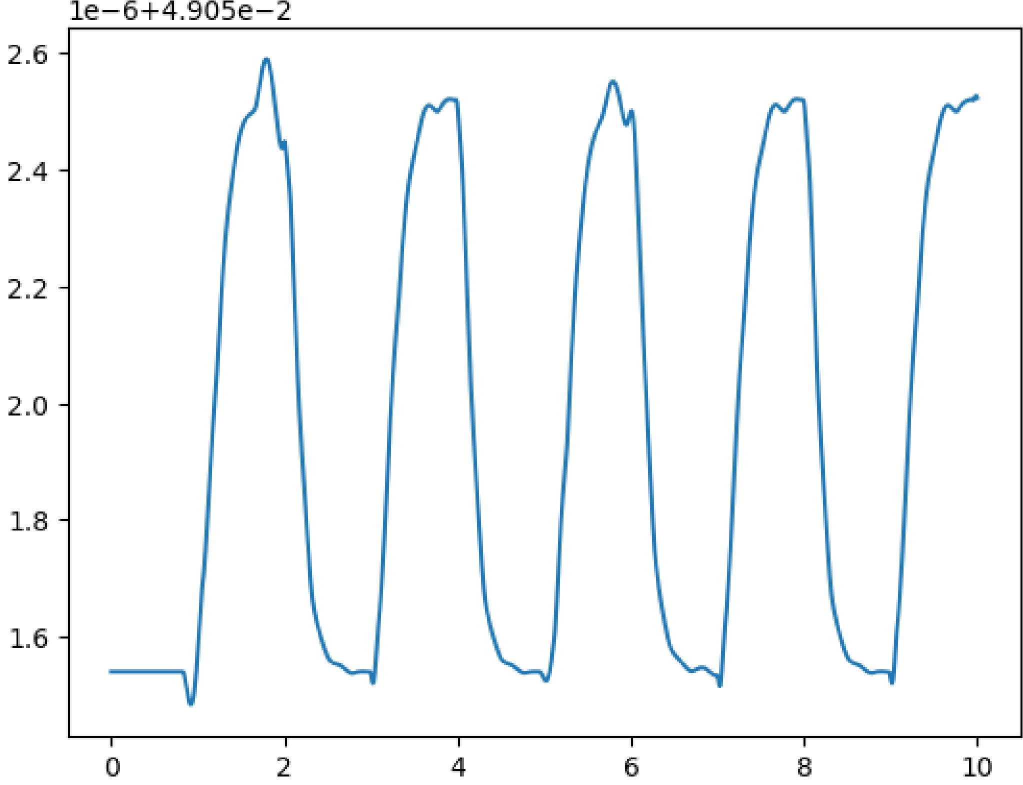
(continues on next page)

(continued from previous page)

```
print(sensor_time_pulse)

#1000 samples over 1 microsecond
end_time = 10
num_pts = 1000
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 2, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0), time_
↪dependence: <function pulse_field at 0x000002333D673600>, coherent_cc: 1.0,␣
↪label: (2,3)}
Decoherent Couplings:
    (0,0): {gamma_transit: 10.0}
    (1,0): {gamma_transit: 10.0}
    (2,0): {gamma_transit: 10.0}
    (2,1): {gamma: 1.0}
    (3,0): {gamma_transit: 10.0}
    (3,2): {gamma: 1.0}
Energy Shifts:
    None
```

```
sol_pulse = rq.solve_time(sensor_time_pulse, end_time, num_pts)
susc_pulse = sol_pulse.coupling_coefficient_observable()

fig, ax = plt.subplots()
ax.plot(sol_pulse.t, susc_pulse.imag);
plt.show()
```



We did not leave enough time to enter a steady-state on any particular pulse (again, we are not trying to demonstrate meaningful physics in this notebook), but we can see that the solver ran fine, and the plusing behavior we would expect is present.

3.5.4 3.4 Multi-value Parameters in a time solve

Just like `solve_steady_state`, `solve_time` supports coupling parameter definitions that are list-like. It works more or less as you might expect based on the behavior of steady-state solving, but it is worth briefly discussing for the sake of demonstrating what the solution might look like.

We will start with a `Sensor` identical to the one in 3.3, but with the `(2,3)` transition's `rabi_frequency` and `detuning` defined as lists. Note that because we are scanning 2 parameters over 51 values, there are actually $51 \times 51 = 2,601$ sets of equations being solved. `rydiqule` is pretty efficient, even in the time solver, for reasonable frequencies, but is not magical. This cell may take longer than previous ones (and probably spin up your computer fans a little), but it should be doable.

```
#101 Mrad/s field, modulated
def my_field_rwa(t):
    return np.sin(10*t)

dets = np.linspace(-5,5,51)
rabis = np.linspace(-1,1,51)

sensor_time_scan = rq.Sensor(4)
sensor_time_scan.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_scan.add_coupling((1,2), detuning=0, rabi_frequency=2)
sensor_time_scan.add_coupling((2,3), detuning=dets, rabi_frequency=rabis, time_
↪dependence=my_field_rwa)

sensor_time_scan.add_transit_broadening(10)
sensor_time_scan.add_decoherence((2,1), 1)
sensor_time_scan.add_decoherence((3,2), 1)
```

(continues on next page)

(continued from previous page)

```
print(sensor_time_scan)

#1000 samples over 3 microsecond
end_time = 3
num_pts = 1000

sol_time_scan =  rq.solve_time(sensor_time_scan, end_time, num_pts, atol=1e-9)

susc_scan = sol_time_scan.coupling_coefficient_observable()
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 2, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: <parameter with 51 values>, detuning: <parameter with␣
↪51 values>, phase: 0, kvec: (0, 0, 0), time_dependence: <function my_field_rwa␣
↪at 0x000002333B4BD9E0>, coherent_cc: 1.0, label: (2,3)}
Decoherent Couplings:
    (0,0): {gamma_transit: 10.0}
    (1,0): {gamma_transit: 10.0}
    (2,0): {gamma_transit: 10.0}
    (2,1): {gamma: 1.0}
    (3,0): {gamma_transit: 10.0}
    (3,2): {gamma: 1.0}
Energy Shifts:
    None
```

As a note, we tightened our absolute tolerance on the time solver. Why is somewhat beyond the scope of this notebook, but it is worth briefly explaining. Under the hood, `rydiqule` uses `scipy`'s `solve_ivp` function with a Runge-Kutta method of order 4(5). To make this work seamlessly with large stacks, `rydiqule` performs some reshape operations, and with such a large vector, the `"RK45"` hueristics will behave differently on an huge list of values as opposed to just the 15 in a single density matrix.

The details are not relevant here, but these sorts of numerical analyses are something `rydiqule` can only take so far, you ultimately need to judge whether a potential numerical error has occured. We made the above choice by analyzing what the plot looked like. As a general rule, if the plot is behaving unexpectedly, the `atol` can be set an order of magnitude or so smaller than any oscillations. You may have astutely noticed that the RWA and non RWA examples above did not match *exaclty*, and this was almost certainly the reason. However, we made the decision not to adjust since we caputred the relevant behavior just fine. Ultimately, `rydiqule` uses the `scipy` default of 1e-3, but you may need to adjust it to suit your problem, just like you would with any ivp solver.

Anyhow, just as we did before, let us investigate some key parts of the `Solution` `sol_time_scan`:

```
print(sol_time_scan.rho.shape)
print(sol_time_scan.axis_labels)
```

```
(51, 51, 1000, 15)
['(2,3)_detuning', '(2,3)_rabi_frequency', 'time', 'density_matrix']
```

So we can see that the shape is as expected. The ordering of the time axes here is general – in a `rydiqule` time soluion, axes will always be ordered as parameters, time, density matrix.

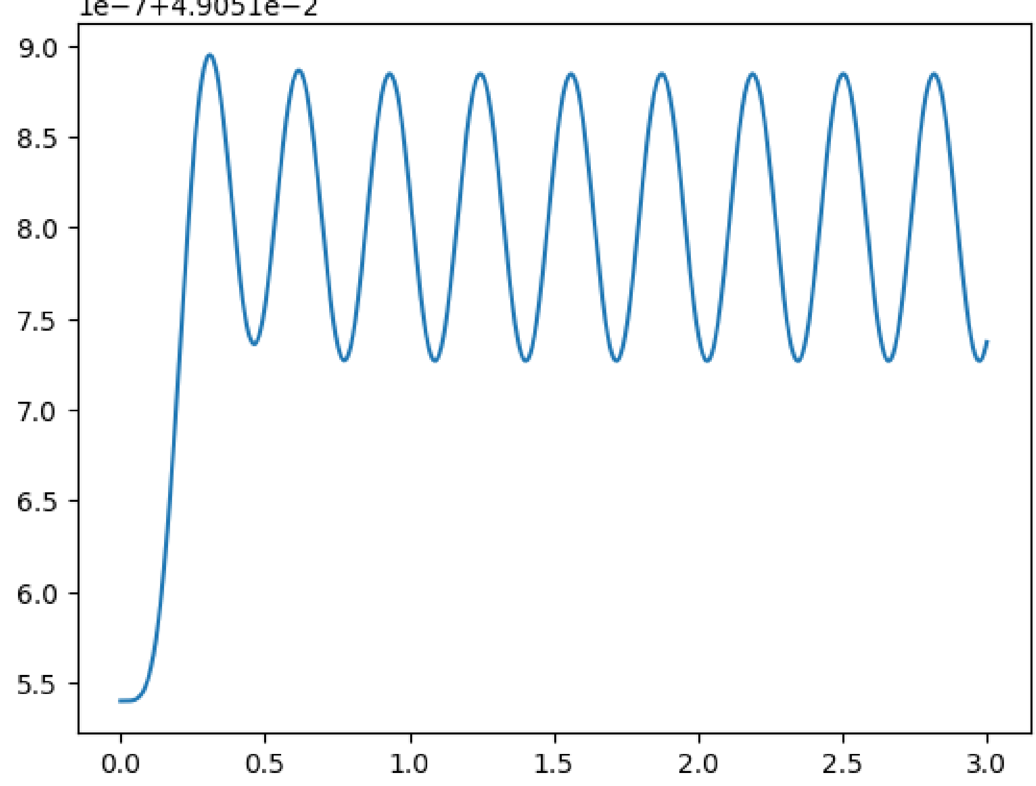
Just for fun, lets try and plot the solution, that corresponds to the single-value solve above, at `detuning=1` and `rabi_frequency=1`. We use `np.where()` to get the correct indeces. Note that we can still apply functions like

observales to the solution without changing the indexing

```
det_idx = np.where(dets==1)[0][0]
rabi_idx = np.where(rabis==1)[0][0] #the return typle of where is tuple(array),␣
↪although there is only 1 index

time = sol_time_scan.t
print(susc_scan.shape)
susc_plot = susc_scan[30,50] #apply squeeze operatiob because indexing leaves
```

```
(51, 51, 1000)
```

```
fig, ax = plt.subplots()
ax.plot(time, susc_plot.imag)
plt.show()
```



3.5.5 3.5 Compiled timesolver backend

With version 2.2.0, rydiqule introduced a fully compiled timesolver backend: `cyrk_solve`. This backend is written in cython and leverages CyRK's `cysolve_ivp`. It is generally faster than the default scipy backend by around an order of magntiude, but requires that the time-dependence functions be compiled with cython by hand (including having a local C compiler). No other aspect of `Sensor` creation or time-solving changes, and the time-function compilation is readily achieved in a jupyter notebook, as will be shown below.

Further details on `cyrk_solve` usage are in the API Docs. A detailed exploration of the performance relative to the scipy backend is located in an example notebook.

Note that the following three cells require that an appropriate C/C++ compiler and OpenMP library is installed on the system.

```
%load_ext Cython
```

```
%%cython
# cython: initializedcheck=False, cdivision=True, nonecheck=False,␣
↪boundscheck=False, wraparound=False

from libc.math cimport sin
# have to use complex math functions from here to avoid MSVC C99 shenanigans
from rydiqule.stack_solvers.cyrk_solver.complex cimport double_complex
from rydiqule.stack_solvers.cyrk_solver cimport tfunc_wrapper

cdef double complex my_field_rwa_cy(double t) noexcept nogil:
    cdef double s = sin(10*t)
    return double_complex(s, 0.0) # must use double_complex(real, imag) to define␣
↪complex numbers

wrapped_rwa_cy = tfunc_wrapper(my_field_rwa_cy)
```

```
sensor_time_rwa_cy = rq.Sensor(4)
sensor_time_rwa_cy.add_coupling((0,1), detuning=0, rabi_frequency=1)
sensor_time_rwa_cy.add_coupling((1,2), detuning=0, rabi_frequency=2)
sensor_time_rwa_cy.add_coupling((2,3), detuning=1, rabi_frequency=1, time_
↪dependence=wrapped_rwa_cy)

sensor_time_rwa_cy.add_transit_broadening(10)
sensor_time_rwa_cy.add_decoherence((2,1), 1)
sensor_time_rwa_cy.add_decoherence((3,2), 1)

print(sensor_time_rwa_cy)

#1000 samples over 3 microsecond
end_time = 3
num_pts = 1000

sol_rwa_cy = rq.solve_time(sensor_time_rwa_cy, end_time, num_pts, solver='cyrk')
```

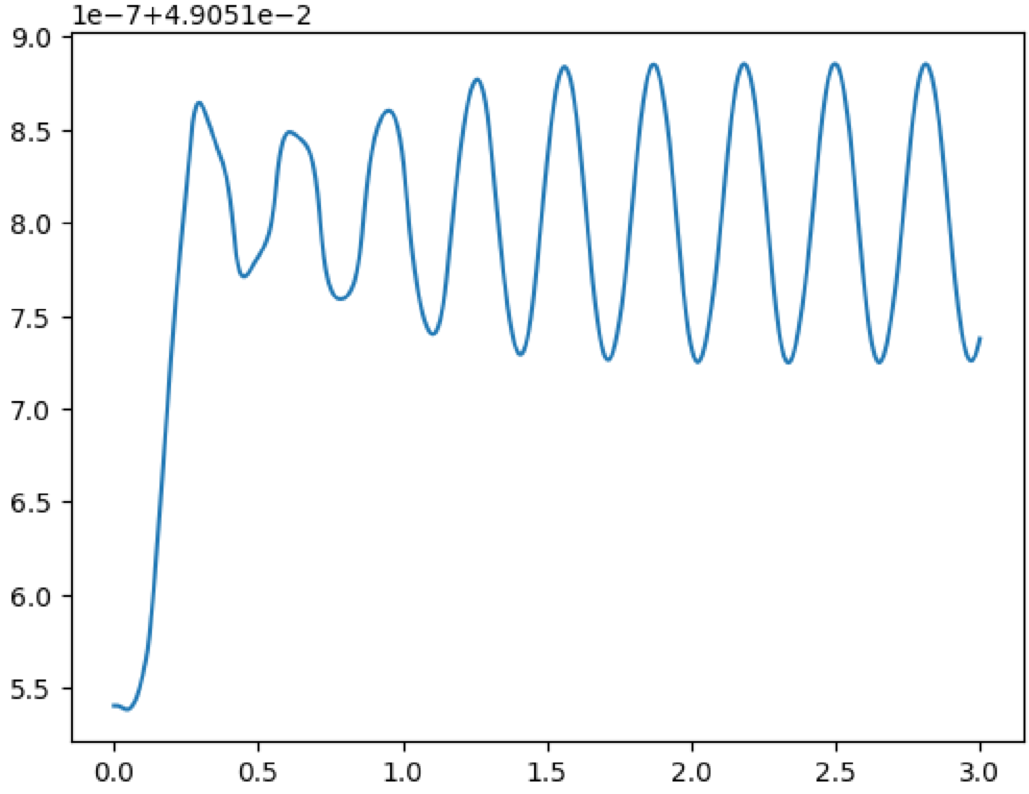
```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 2, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0), time_
↪dependence: <rydiqule.stack_solvers.cyrk_solver.utils.Tfunc_Wrapper object at␣
↪0x0000023353ECF1D0>, coherent_cc: 1.0, label: (2,3)}
Decoherent Couplings:
    (0,0): {gamma_transit: 10.0}
    (1,0): {gamma_transit: 10.0}
    (2,0): {gamma_transit: 10.0}
    (2,1): {gamma: 1.0}
    (3,0): {gamma_transit: 10.0}
    (3,2): {gamma: 1.0}
Energy Shifts:
    None
```

```
fig, ax = plt.subplots()
susc=sol_rwa_cy.coupling_coefficient_observable()
```

(continues on next page)

(continued from previous page)

```
ax.plot(sol_rwa_cy.t, susc.imag);
plt.show()
```



3.6 4. Simulating Doppler broadening

One final key piece of functionality in `rydiqule` is the ability to simulate a doppler-broadened system. Internally, this is handled by breaking a doppler velocity profile into some number of slices, then applying the corresponding detunings for each class to the unshifted solution. This produces many sets of equations of motion, which are then solved, and then a weighted average performed to get the doppler-broadened solution.

To set up a system with doppler broadening, we use the the `"kvec"` keyword the couplings definition along with the `vP` attribute. A `kvec` is a 3-element spatial vector corresponding to the usual definition of a k-vector for an electromagnetic plane wave, defined in units of Mrad/m. `Sensor.vP` is an attribute that defines the most probable speed for an atom in a given sensor assuming a 3-D Maxwell-Boltzmann velocity distribution. Note that when multiplied `vP` and `kvec` correspond to the most probable doppler shift vector with respect to a given laser in Mrad/s.

Configuring doppler settings can get involved, and much of the functionality surrounding doppler handling is beyond the scope of an introductory notebook. This will be a quick discussion, but should at least be useful in handling most simple cases. If applying the steps outlined here produces unexpected results, you can find more details about how `rydiqule` does doppler in the physics and api doppler documentation.

Note that as of version 2.1.0, in addition to `solve_steady_state` below, rydiqule supports analytic doppler averaging. More details of this function can be found in physics and api documentation.

3.6.1 4.1 A simple doppler example.

Let us recreate the system in 2.4 above, which was the first thing we plotted. Recall that there was a large main peak, consisting of 3 sharp subpeaks. If we account for a small amount of doppler, broadening, we would expect those sharper features to be smeared out. In the example below we will apply doppler broadening to a pair of co-propagating lasers.

A doppler broadening of 1 MHz is wider than the narrow subpeaks but narrower than the main peak, so with that amount of broadening, the large peak should stay similar in width, but the small peaks will no longer be able to be resolved. Let us test this hypothesis as a quick smell-check that `rydiqule` is doing something reasonable with doppler broadening.

Recall that `kvec` is an attribute of a particular coupling, while `vP` is a property of sensor, and can be defined either in the constructor using `Sensor(....,vP=<value>)`, or after construction with `my_sensor.vP = <value>`

```
basis_size = 4
sensor_doppler = rq.Sensor(basis_size, vP=2)

k_mag = np.pi #Mrad/m
k_direction = np.array([1,0,0])

detunings = 2*np.pi*np.linspace(-10,10,201) #201 values between -10 and 10 MHz
probe = {"states":(0,1), "detuning": detunings, "rabi_frequency": 3, 'kvec': k_
↪direction*k_mag}
coupling = {"states":(1,2), "detuning": 0, "rabi_frequency": 5, 'kvec': k_
↪direction*k_mag}
rf = {"states":(2,3), "detuning": 0, "rabi_frequency":7}

sensor_doppler.add_couplings(probe, coupling, rf)

gamma = np.zeros((basis_size, basis_size))
gamma[1:,0] = 0.1
sensor_doppler.set_gamma_matrix(gamma)

print(sensor_doppler)
```

```
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 3, detuning: <parameter with 201 values>, phase: 0,␣
↪kvec: <parameter with 3 values>, coherent_cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 5, detuning: 0, phase: 0, kvec: <parameter with 3␣
↪values>, coherent_cc: 1.0, label: (1,2)}
    (2,3): {rabi_frequency: 7, detuning: 0, phase: 0, kvec: (0, 0, 0), coherent_
↪cc: 1.0, label: (2,3)}
Decoherent Couplings:
    (1,0): {gamma: 0.1}
    (2,0): {gamma: 0.1}
    (3,0): {gamma: 0.1}
Energy Shifts:
    None
```

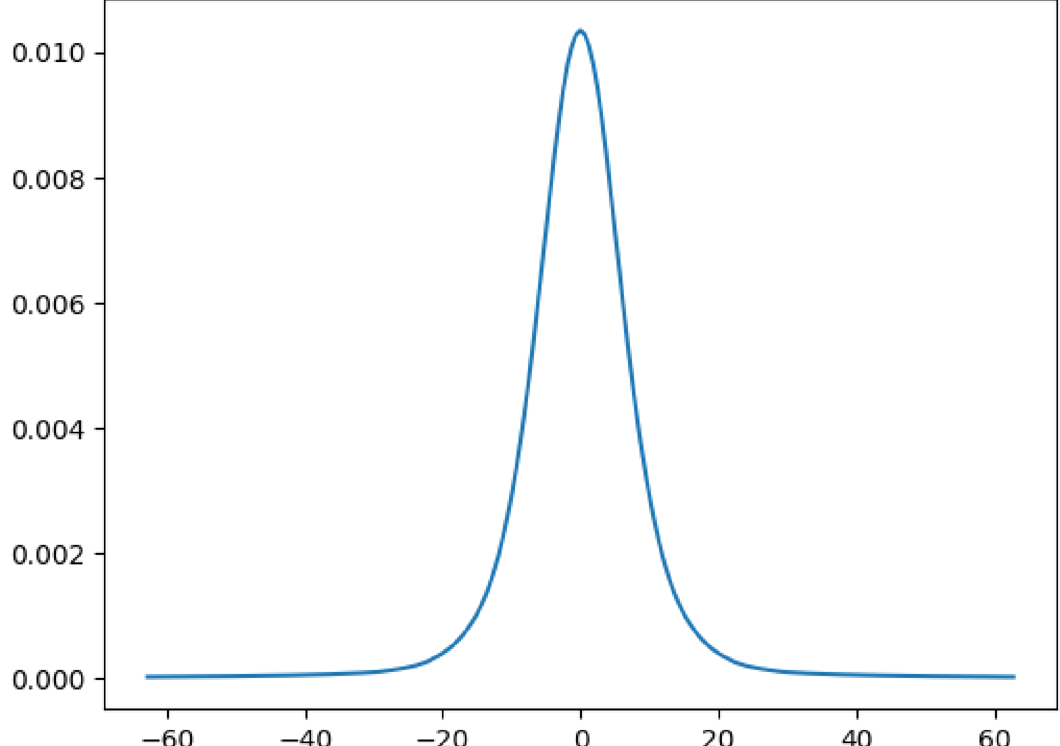
Before checking out the solution, Let us briefly touch on what is happening under the hood with a doppler solve. `rydiqule` leverages its existing stack framework internally by adding more hamiltonians to the stack. It will create and solve a set of Hamiltonians and equations for each doppler class. Then at the end, it will apply weights based on the population in each class assuming a Boltzmann distribution of velocity classes, and sum the weighted classes together. Note that the solution's final shape is the same as it would be without doppler. So the internals of dopper are, in basic cases, something you don't actually need to see.

Note that since doppler is a parameter of the solve rather than an argument of `Sensor` itself,

```
solution_doppler = rq.solve_steady_state(sensor_doppler, doppler=True)
print(f"Solution shape: {solution_doppler.rho.shape}")
```

```
Solution shape: (201, 15)
```

```
absorption_doppler = solution_doppler.coupling_coefficient_observable().imag
plt.plot(detunings, absorption_doppler)
plt.show()
```



Perfect, it is as we expected: The three narrow peaks comprising the larger one have been blurred into a single peak.

It is also worth mentioning that the same argument works identically in the `solve_time()` function.

3.6.2 4.2 A word of caution

Once you realize `rydiqule` can do doppler broadening, it may be tempting to apply it to all your experiments. It is important to keep in mind that for each axis over which you account for doppler broadening, you are increasing the number of equations to solve by a factor of $n_{classes}$, which by default is ~600. Remember, in its current iteration `rydiqule` explicitly solves every doppler class, it does not use any hueristics. Adding doppler broadening can quickly get out of control in both memory and computation time. A seemingly innocuous `kvec=k_mag*(1,1,1)` will apply doppler broadening in 3 dimensions and increase the number of equations `rydiqule` solves by a factor of 8,000,000. This is not to tell you not to use doppler, just to understand what you are asking `rydiqule` to do.

3.7 5. What Next?

Hopefully, you now have a good idea of the basics of how `rydiqule` can be used to solve atomic systems in a variety of ways. Depending what you want to do, you can start running your own simulations.

If you intend to simulate real-world lab experiments with actual Rydberg atoms, the `rydiqule.Cell` module has tools to do exaclty that. It inherits from `Sensor`, so all the basic principles apply, but some funcitonality has been added to calulate things like dipole moments, decay rates, and state energies automatically using functionality from the `ARC` package. We have a `Cell_Basics.ipynb` notebook in the same folder as this one, and you now equipped to go through that tutorial. That notebook *will* assume that you understand the principles of this one, but it provides useful tools to start simulating your own Rubidium. After that, the rest of the examples in this notebook show examples using `Cell` to simulate real experiments that you could do in a lab.

CHAPTER

FOUR

INTRODUCTION TO CELL AND REAL ATOMS

The graph-based framework that rydiqule is based on is incredibly flexible, but often flexibilty is not as useful as being able to perform common actions quickly. For example, the Sensor module of rydiqule allows for any value of any parameter to be supported. While this is great for flexibility, if you know the first two states of an atomic system are all m_j fine structrue states for the D2 line of Rubidium-87, it would be somewhat painful to manually specify the dipole moment and transition frequencies for every possible state pairing for the laser coupling the the two manifolds.

The Cell module of rydiqule is designed to address exactly this issue. It builds upon Sensor with additional functionality to automatically add quantities like quantum numbers, dipole moments, state energies, and decoherence rates. Indeed, if you are familiar with object-oriented programming in python, Cell in fact inherits Sensor as a subclass, meaning it has access to all the same methods, and has the same internal underlying structure. Importantly, Cell is *specifically* designed to model alkali atoms supported by the Alkali Rydberg Calculator project.

This notebook can be downloaded here.

4.1 0. What is a Cell?

As we alluded to above, a Cell is just a Sensor at the end of the day. The familiar functions like add_coupling and get_hamiltonian are still there, but there is a lot more under the hood.

This notebook will go over some of the basic ways to use Cell to solve real systems quickly and easily. This tutorial assumes you have gone through and understand the basics laid out in the Introduction_To_Rydiqule notebook. If any of the terminolgy in this notebook is unfamiliar, revisit that notebook for a demonstration of basic principles. This notebook will not revisit those basic priciples, it will primarily highlight the differences between Cell and Sensor.

This notebook will also assume a basic familiarity with rydberg atom atomic physics. While we do have some physics documentation, rydiqule is aimed primarily at physicists who already understand many of these concepts, and the docs are more for our implementation of the underlying physics.

For starters, we will again import rydiqule as rq, as well as the usual imports. We also import A_QState directly for reasons that will become clear momentarily.

```
import rydiqule as rq
from rydiqule import A_QState

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

4.2 1. Creating a Cell

4.2.1 1.1 A_QState and constructor basics

If you remember in Sensor, the states could be integers, strings, or tuples. Cell is a little bit more restrictive about what you can use for states. To describe states in Cell, rydiqule has introduced a new type using python's

namedtuple called A_QState, short for atomic quantum state. Note that this is a subclass of tuple, so no rules have changed; using this is a special case of the bases that can be defined in Sensor. This named tuple has 3 mandatory arguments, n, l, and j representing the first 3 quantum number of a rydberg quantum state. Additionally, there are 3 more optional arguments for m_j, f, and m_f. As you might expect, you cannot specify f or m_f if m_j is specified, or vice versa. Also, to specify m_f, f must also be specified. All the usual quantum number rules for rydberg atoms must be obeyed.

We can access an A_QState directly as rq.A_QState, but it can be easier to from rydiqule import A_QState to shorten, since you may need to call it often. Let's start with a simple ground and excited state for the D1 line of Rubidium-85 just to see how the Cell constructor works. As a minimum, we need to specify a atom with a string, and pass a list of A_QStates.

```
g = A_QState(5, 0, 0.5)
e = A_QState(5, 1, 0.5)

Rb_Cell = rq.Cell("Rb85", [g, e])
print(Rb_Cell.states)
print(type(Rb_Cell.states[0]))
```

```
[(n=5, l=0, j=0.5), (n=5, l=1, j=0.5)]
<class 'rydiqule.atom_utils.A_QState'>
```

The atom string can be any atoms supported by the ARC package, and following the <atomic symbol><isotope number> as above. Consult arc to see what isotopes are supported.

For readablity, A_QState trims off the "A_QState" text and unused quantum numbers from the string output, but rest assured, it is still an A_QState, which we see in the printout. You can see that the states of the system are exactly what are in the list we passed to the constructor. With a very simple system defined, we can inspect the Cell.couplings graph to see what sorts of things are on the graph.

```
print(Rb_Cell.couplings.nodes(data=True))
print(Rb_Cell.couplings.edges(data=True))
```

```
[((n=5, l=0, j=0.5), {'energy': 0.0, 'gamma_lifetime': 0.0}), ((n=5, l=1, j=0.5), {
↪'energy': 2369435883.882498, 'gamma_lifetime': 36.11450417508357})]
[((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), {'gamma_transition': 36.11450417508357,
↪'label': '((5, 1, 0.5),(5, 0, 0.5))'})]
```

We can see that a couple of things have been added to the graph. Each node has been populated automatically with energy levels (with the ground state defined at 0 Mrad), and state lifetimes. Like basically everything else in rydiqule, these quantities are expressed in Mrad/s. In addition to data on the nodes (which is just for reference and does not affect solving directly), there is a decoherent transition added from the second state to the first, associated with natural state lifetime of the $5P_{1/2}$ state before it decays back to the ground state. In general, rydiqule will add natural transition rates from higher to lower states. There are some caveats and details to how this is done that will be discussed later in this section. Just for fun, let's demonstrate that even without anything else explicitly defined, we already have a decoherence_matrix.

```
print(Rb_Cell.decoherence_matrix())
```

```
[[ 0.          0.        ]
 [36.11450418  0.        ]]
```

4.2.2 1.2 Efficiently defining A_QStates

List shorthand

While it is nice that we can define all of these states with A_QState, it is easy to imagine a situation in which there are a lot more states in a system of interest that you want to write down (consider all possible hyperfine sublevels in a high angular momentum upper rydberg state). You could be defining the states for hours. As you might imagine,

`rydiqule` has a built-in way to define all of these states at once. When we define our `A_QState`, we can define any of the quantum numbers from `j` onwards (so `m_j`, `f`, `m_f`) as a list, expanding the single specification out into a list of states automatically. Below we show this shorthand to add both the D1 and D2 excited states with one `A_QState` specification. This functionality is identical to using `tuple` states in a `Sensor`, and indeed is the motivation for adding the functionality in the first place.

```
g = A_QState(5, 0, 0.5)
e = A_QState(5, 1, [0.5, 1.5])

Rb_Cell_D12 = rq.Cell("Rb85", [g, e])
print(Rb_Cell_D12.states)
print(Rb_Cell_D12.decoherence_matrix())
```

```
[(n=5, l=0, j=0.5), (n=5, l=1, j=0.5), (n=5, l=1, j=1.5)]
[[ 0.          0.          0.        ]
 [36.11450418  0.          0.        ]
 [38.11316014  0.          0.        ]]
```

We can see that again we have created a `Cell` with decoherence values from the natural state lifetime already added, and once again we get the `decoherence_matrix` of the system (with no additional broadening) automatically.

"All" shorthand

Even providing things as lists can get cumbersome once sublevels start to get involved. For this reason, `rydiqule` supports using the string `"all"` for quantum numbers rather than just a list, which will automatically get states containing all allowed values of the specified quantum numbers. Let us again consider just the D2 line, but account for the `m_j` splitting of these levels.

```
g = A_QState(5, 0, 0.5, m_j="all")
e = A_QState(5, 1, 1.5, m_j="all")

D2_Cell_mj = rq.Cell("Rb85", [g,e])
print(D2_Cell_mj.states)
print(D2_Cell_mj.decoherence_matrix())
```

```
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), (n=5, l=1, j=1.5, m_j=-1.
↪5), (n=5, l=1, j=1.5, m_j=-0.5), (n=5, l=1, j=1.5, m_j=0.5), (n=5, l=1, j=1.5, m_
↪j=1.5)]
[[ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [38.11316014  0.          0.          0.          0.          0.        ]
 [25.40877343 12.70438671  0.          0.          0.          0.        ]
 [12.70438671 25.40877343  0.          0.          0.          0.        ]
 [ 0.         38.11316014  0.          0.          0.          0.        ]]
```

We can see that we have 2 states in the ground manifold (corresponding to $m_j = \pm 0.5$), and 4 in the excited manifold (corresponding to $m_j = \pm 0.5, \pm 1.5$). Once again, we have sucessfully accounted for the transitions from excited into the ground state, including zeros for transitions which are not dipole-allowed.

We can take this `"all"` notion one step further by defining an entire hyperfine manifold. This time we will account for hyperfine splitting in both the ground and excited state. Note when computing hyperfine splitting, since we supplied the atom specification to the constructor, `rydiqule` already knows the nuclear magnetic moment of our atom (Rubidium-85) to be $\frac{5}{2}$, and will calculate allowed `f` values accordingly

```
g = A_QState(5, 0, 0.5, f="all", m_f="all")
e = A_QState(5, 1, 1.5, f="all", m_f="all")

D2_Cell_hyperfine = rq.Cell("Rb85", [g,e])
```

(continues on next page)

(continued from previous page)

```
print(D2_Cell_hyperfine.states)
print()
print(f"{len(D2_Cell_hyperfine.states)} states")
```

```
[(n=5, l=0, j=0.5, f=2.0, m_f=-2.0), (n=5, l=0, j=0.5, f=2.0, m_f=-1.0), (n=5, l=0,
↪ j=0.5, f=2.0, m_f=0.0), (n=5, l=0, j=0.5, f=2.0, m_f=1.0), (n=5, l=0, j=0.5,␣
↪f=2.0, m_f=2.0), (n=5, l=0, j=0.5, f=3.0, m_f=-3.0), (n=5, l=0, j=0.5, f=3.0, m_
↪f=-2.0), (n=5, l=0, j=0.5, f=3.0, m_f=-1.0), (n=5, l=0, j=0.5, f=3.0, m_f=0.0),␣
↪(n=5, l=0, j=0.5, f=3.0, m_f=1.0), (n=5, l=0, j=0.5, f=3.0, m_f=2.0), (n=5, l=0,␣
↪j=0.5, f=3.0, m_f=3.0), (n=5, l=1, j=1.5, f=1.0, m_f=-1.0), (n=5, l=1, j=1.5,␣
↪f=1.0, m_f=0.0), (n=5, l=1, j=1.5, f=1.0, m_f=1.0), (n=5, l=1, j=1.5, f=2.0, m_
↪f=-2.0), (n=5, l=1, j=1.5, f=2.0, m_f=-1.0), (n=5, l=1, j=1.5, f=2.0, m_f=0.0),␣
↪(n=5, l=1, j=1.5, f=2.0, m_f=1.0), (n=5, l=1, j=1.5, f=2.0, m_f=2.0), (n=5, l=1,␣
↪j=1.5, f=3.0, m_f=-3.0), (n=5, l=1, j=1.5, f=3.0, m_f=-2.0), (n=5, l=1, j=1.5,␣
↪f=3.0, m_f=-1.0), (n=5, l=1, j=1.5, f=3.0, m_f=0.0), (n=5, l=1, j=1.5, f=3.0, m_
↪f=1.0), (n=5, l=1, j=1.5, f=3.0, m_f=2.0), (n=5, l=1, j=1.5, f=3.0, m_f=3.0),␣
↪(n=5, l=1, j=1.5, f=4.0, m_f=-4.0), (n=5, l=1, j=1.5, f=4.0, m_f=-3.0), (n=5,␣
↪l=1, j=1.5, f=4.0, m_f=-2.0), (n=5, l=1, j=1.5, f=4.0, m_f=-1.0), (n=5, l=1, j=1.
↪5, f=4.0, m_f=0.0), (n=5, l=1, j=1.5, f=4.0, m_f=1.0), (n=5, l=1, j=1.5, f=4.0,␣
↪m_f=2.0), (n=5, l=1, j=1.5, f=4.0, m_f=3.0), (n=5, l=1, j=1.5, f=4.0, m_f=4.0)]

36 states
```

Obviously, we get a lot of states when we account for hyperfine splitting. While we do show how easy something like this, this is also somewhat of a warning. On the one hand, it is easy to account for tons of states with just a couple lines. On the other hand, a larger basis will take longer to solve (only in polynomial time, but still longer). This is not to discourage you from using the `"all"` feature, but to point out that you should only do so when you are looking to model physics only available when accounting for hyperfine. A couple of notes about some rules of hyperfine splitting:

1. We do not support mixing `n,l,j` states with hyperfine or fine states in the same `Cell`. If you want to use hyperfine, all states must be either fine split or hyperfine split.
2. `A_QStates` cannot include `f` without `m_f`. If you want hyperfine splitting and specify `f`, `m_f` must be a single value, list of allowed values, or `"all"`. In any case, `rydiqule` will enforce that these quantum numbers are physically allowed.

Helper function shorthand

This is less crucial, but one more way `rydiqule` provides to specify manifolds of states in `Cell` is with a handful of utility functions that return lists of states. These can be useful if you either can't be bothered to type the full `A_QState` out yourself or if you want to test a system with multiple different atoms and only change a single value in code. The relevant functions are as follows. In each case, `n` can be either an integer or a string of a particular atom as specified in the constructor (eg `"Rb85"`). `splitting` is one of `[None, "fs", or "hfs"]`, with the default being `None`:

1. `rq.ground_state(n, splitting=...)`, gets the $l = 0, j = 0.5$ state(s). This is what the in the "energy" value on the nodes seen previouslt are in reference to.
2. `rq.D1_excited(n, splitting=...)`, gets the $l = 1, j = 0.5$ state(s).
3. `rq.D2_excited(n, splitting=...)`, gets the $l = 1, j = 1.5$ state(s).
4. `rq.D1_states(n, splitting=..., g_splitting=..., e_splitting=..)` just calls 1 and 2.`splitting` overrrides ground and excited splitting if present.
5. `rq.D2_states(n, splitting=..., g_splitting=..., e_splitting=..)` just calls 1 and 2. `splitting` overrrides ground and excited splitting if present.

Here we can see these in action for the D1 states:

```
atom = "Cs"
g = rq.ground_state(atom, splitting="fs")
e = rq.D1_excited(atom, splitting="hfs")

Cs_cell = rq.Cell(atom, [g,e])
print(Cs_cell.states)
```

```
[(n=6, l=0, j=0.5, m_j=-0.5), (n=6, l=0, j=0.5, m_j=0.5), (n=6, l=1, j=0.5, f=3.0,␣
↪m_f=-3.0), (n=6, l=1, j=0.5, f=3.0, m_f=-2.0), (n=6, l=1, j=0.5, f=3.0, m_f=-1.
↪0), (n=6, l=1, j=0.5, f=3.0, m_f=0.0), (n=6, l=1, j=0.5, f=3.0, m_f=1.0), (n=6,␣
↪l=1, j=0.5, f=3.0, m_f=2.0), (n=6, l=1, j=0.5, f=3.0, m_f=3.0), (n=6, l=1, j=0.5,
↪ f=4.0, m_f=-4.0), (n=6, l=1, j=0.5, f=4.0, m_f=-3.0), (n=6, l=1, j=0.5, f=4.0,␣
↪m_f=-2.0), (n=6, l=1, j=0.5, f=4.0, m_f=-1.0), (n=6, l=1, j=0.5, f=4.0, m_f=0.0),
↪ (n=6, l=1, j=0.5, f=4.0, m_f=1.0), (n=6, l=1, j=0.5, f=4.0, m_f=2.0), (n=6, l=1,
↪ j=0.5, f=4.0, m_f=3.0), (n=6, l=1, j=0.5, f=4.0, m_f=4.0)]
```

Or, equivalently:

```
Cs_cell2 = rq.Cell(atom, rq.D1_states(atom, g_splitting="fs", e_splitting="hfs"))
print(Cs_cell2.states)
```

```
[(n=6, l=0, j=0.5, m_j=-0.5), (n=6, l=0, j=0.5, m_j=0.5), (n=6, l=1, j=0.5, f=3.0,␣
↪m_f=-3.0), (n=6, l=1, j=0.5, f=3.0, m_f=-2.0), (n=6, l=1, j=0.5, f=3.0, m_f=-1.
↪0), (n=6, l=1, j=0.5, f=3.0, m_f=0.0), (n=6, l=1, j=0.5, f=3.0, m_f=1.0), (n=6,␣
↪l=1, j=0.5, f=3.0, m_f=2.0), (n=6, l=1, j=0.5, f=3.0, m_f=3.0), (n=6, l=1, j=0.5,
↪ f=4.0, m_f=-4.0), (n=6, l=1, j=0.5, f=4.0, m_f=-3.0), (n=6, l=1, j=0.5, f=4.0,␣
↪m_f=-2.0), (n=6, l=1, j=0.5, f=4.0, m_f=-1.0), (n=6, l=1, j=0.5, f=4.0, m_f=0.0),
↪ (n=6, l=1, j=0.5, f=4.0, m_f=1.0), (n=6, l=1, j=0.5, f=4.0, m_f=2.0), (n=6, l=1,
↪ j=0.5, f=4.0, m_f=3.0), (n=6, l=1, j=0.5, f=4.0, m_f=4.0)]
```

4.3 2. Decoherence rates in `Cell`

As we can see above, natural decays based on the atom used are added automatically. While the basic mechanics of how decoherence is handled in `Cell` is identical to `Sensor`, there are some differences in what get added to the `couplings` graph that need to be accounted for. In this section we discuss the details of decoherent couplings in the `Cell` class, including what is calculated automatically, what isn't, and how to make it consistent with calculations and experiments outside of `rydiqule`.

4.3.1 2.1 Natural transition rates

Obviously, in an actual atomic vapor, population in higher energies naturally decays to lower energies. In `rydiqule`, this is handled automatically via the `gamma_transition` keyword added to the graph edge. Let us start by recreating the very simple `Cell` we created in section 1 on the D1 line of Rubidium-85, and examine the edges of the graph before anything else gets added.

```
g = A_QState(5, 0, 0.5)
e = A_QState(5, 1, 0.5)

Rb_Cell = rq.Cell("Rb85", [g, e])
print(Rb_Cell.couplings.edges(data=True))
```

```
[((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), {'gamma_transition': 36.11450417508357,
↪'label': '((5, 1, 0.5),(5, 0, 0.5))'})]
```

Here we can indeed see that, unlike in a basic `Sensor`, the transition rate (in Mrad/s as always) is added from the excited state to the ground state without any extra work on our part. This will be the case no matter how many states

are added to the `Cell`. Let us again examine this attribute but with the full hyperfine manifolds of the D1 line.

```
atom = "Cs" #Cesium for fun, only one isotope (133) supported by ARC so the␣
↪isotope number is omitted

D1_hfs_cell = rq.Cell(atom, rq.D1_states(atom, splitting="hfs"))
for s1, s2, value in D1_hfs_cell.couplings.edges(data="gamma_transition"):
    print(f"({s1},{s2}):{value}")
```

```
((6, 1, 0.5, f=3.0, m_f=-3.0),(6, 0, 0.5, f=3.0, m_f=-3.0)):5.37334121162191
((6, 1, 0.5, f=3.0, m_f=-3.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):1.7911137372073023
((6, 1, 0.5, f=3.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-4.0)):16.717061547268166
((6, 1, 0.5, f=3.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-3.0)):4.1792653868170415
((6, 1, 0.5, f=3.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):0.5970379124024344
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-3.0)):1.7911137372073023
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):2.388151649609737
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):2.9851895620121716
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-3.0)):12.537796160451125
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):7.16445494882921
((6, 1, 0.5, f=3.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):1.7911137372073025
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):2.9851895620121702
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):0.5970379124024342
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=0.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):8.955568686036512
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):8.955568686036512
((6, 1, 0.5, f=3.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=0.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=0.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=0.0),(6, 0, 0.5, f=3.0, m_f=1.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=0.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):5.970379124024342
((6, 1, 0.5, f=3.0, m_f=0.0),(6, 0, 0.5, f=4.0, m_f=0.0)):9.55260659843895
((6, 1, 0.5, f=3.0, m_f=0.0),(6, 0, 0.5, f=4.0, m_f=1.0)):5.970379124024342
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=0.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=1.0)):0.5970379124024342
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=2.0)):2.9851895620121702
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=0.0)):3.5822274744146045
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=1.0)):8.955568686036512
((6, 1, 0.5, f=3.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=2.0)):8.955568686036512
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=1.0)):2.9851895620121716
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=2.0)):2.388151649609737
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=3.0)):1.7911137372073023
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=1.0)):1.7911137372073025
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=2.0)):7.16445494882921
((6, 1, 0.5, f=3.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=3.0)):12.537796160451125
((6, 1, 0.5, f=3.0, m_f=3.0),(6, 0, 0.5, f=3.0, m_f=2.0)):1.7911137372073023
((6, 1, 0.5, f=3.0, m_f=3.0),(6, 0, 0.5, f=3.0, m_f=3.0)):5.37334121162191
((6, 1, 0.5, f=3.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=2.0)):0.5970379124024344
((6, 1, 0.5, f=3.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=3.0)):4.1792653868170415
((6, 1, 0.5, f=3.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=4.0)):16.717061547268166
((6, 1, 0.5, f=4.0, m_f=-4.0),(6, 0, 0.5, f=3.0, m_f=-3.0)):16.71706154726816
((6, 1, 0.5, f=4.0, m_f=-4.0),(6, 0, 0.5, f=4.0, m_f=-4.0)):9.552606598438953
((6, 1, 0.5, f=4.0, m_f=-4.0),(6, 0, 0.5, f=4.0, m_f=-3.0)):2.388151649609738
((6, 1, 0.5, f=4.0, m_f=-3.0),(6, 0, 0.5, f=3.0, m_f=-3.0)):4.17926538681704
((6, 1, 0.5, f=4.0, m_f=-3.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):12.537796160451125
((6, 1, 0.5, f=4.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-4.0)):2.388151649609738
((6, 1, 0.5, f=4.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-3.0)):5.37334121162191
((6, 1, 0.5, f=4.0, m_f=-3.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):4.179265386817042
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-3.0)):0.5970379124024344
```

(continues on next page)

(continued from previous page)

```
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):7.16445494882921
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):8.955568686036514
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-3.0)):4.179265386817042
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):2.3881516496097372
((6, 1, 0.5, f=4.0, m_f=-2.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):5.373341211621909
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=-2.0)):1.7911137372073025
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):8.955568686036514
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=3.0, m_f=0.0)):5.970379124024343
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=-2.0)):5.373341211621909
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):0.5970379124024343
((6, 1, 0.5, f=4.0, m_f=-1.0),(6, 0, 0.5, f=4.0, m_f=0.0)):5.970379124024346
((6, 1, 0.5, f=4.0, m_f=0.0),(6, 0, 0.5, f=3.0, m_f=-1.0)):3.5822274744146045
((6, 1, 0.5, f=4.0, m_f=0.0),(6, 0, 0.5, f=3.0, m_f=0.0)):9.55260659843895
((6, 1, 0.5, f=4.0, m_f=0.0),(6, 0, 0.5, f=3.0, m_f=1.0)):3.5822274744146045
((6, 1, 0.5, f=4.0, m_f=0.0),(6, 0, 0.5, f=4.0, m_f=-1.0)):5.970379124024342
((6, 1, 0.5, f=4.0, m_f=0.0),(6, 0, 0.5, f=4.0, m_f=1.0)):5.970379124024342
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=0.0)):5.970379124024343
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=1.0)):8.955568686036512
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=3.0, m_f=2.0)):1.7911137372073025
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=0.0)):5.970379124024342
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=1.0)):0.5970379124024343
((6, 1, 0.5, f=4.0, m_f=1.0),(6, 0, 0.5, f=4.0, m_f=2.0)):5.373341211621909
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=1.0)):8.955568686036514
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=2.0)):7.16445494882921
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=3.0, m_f=3.0)):0.5970379124024344
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=1.0)):5.373341211621909
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=2.0)):2.3881516496097372
((6, 1, 0.5, f=4.0, m_f=2.0),(6, 0, 0.5, f=4.0, m_f=3.0)):4.179265386817042
((6, 1, 0.5, f=4.0, m_f=3.0),(6, 0, 0.5, f=3.0, m_f=2.0)):12.537796160451125
((6, 1, 0.5, f=4.0, m_f=3.0),(6, 0, 0.5, f=3.0, m_f=3.0)):4.17926538681704
((6, 1, 0.5, f=4.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=2.0)):4.179265386817042
((6, 1, 0.5, f=4.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=3.0)):5.37334121162191
((6, 1, 0.5, f=4.0, m_f=3.0),(6, 0, 0.5, f=4.0, m_f=4.0)):2.388151649609738
((6, 1, 0.5, f=4.0, m_f=4.0),(6, 0, 0.5, f=3.0, m_f=3.0)):16.71706154726816
((6, 1, 0.5, f=4.0, m_f=4.0),(6, 0, 0.5, f=4.0, m_f=3.0)):2.388151649609738
((6, 1, 0.5, f=4.0, m_f=4.0),(6, 0, 0.5, f=4.0, m_f=4.0)):9.552606598438953
```

That's a lot of couplings! We can already start to see the benefits of using a `Cell` for calculations involving real atoms, since in a `Sensor` these would all need to be added manually. As we mentioned previously, `rydiqule` uses the ARC Rydberg package for all of its under-the-hood calculations in `Cell`, and for more details about how these numbers are computed you can refer to their documentation. It is worth noting that the numbers returned will not be the same since `rydiqule` converts all relevant quanties to Mrad/s for internal consistency.

4.3.2 2.2 Natural state lifetimes

As we saw in example 1.1, the nodes of the `couplings` graph also contain information about the natural lifetime of each state in the `gamma_lifetime` node attribute. We will recreate that example here and see that, for a simple D1 line, the natural lifetime matches the transition rate from the exited to ground state. As a simple sanity check, we also see that the `gamma_lifetime` attribute for the ground state is 0, since population will obviously not decay out of $5S^{1/2}$

```
g = A_QState(5, 0, 0.5)
e = A_QState(5, 1, 0.5)

Rb_Cell = rq.Cell("Rb85", [g, e])
```

(continues on next page)

(continued from previous page)

```
print(Rb_Cell.couplings.nodes(data="gamma_lifetime"))
print(Rb_Cell.couplings.edges(data="gamma_transition"))
```

```
[((n=5, l=0, j=0.5), 0.0), ((n=5, l=1, j=0.5), 36.11450417508357)]
[((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), 36.11450417508357)]
```

This is because there is only an single decay path for an atom in the $5P^{1/2}$ state: straight back to the $5S^{1/2}$ state. However, if we now imagine that the the first excited state is one which has multiple decay paths, it will no longer be the case that the natural state lifetime matches the decay rate. To see this, let us add the $6P^{3/2}$ state as the first excited state:

```
g = A_QState(5, 0, 0.5)
e1 = A_QState(6, 1, 1.5)

Rb_Cell_6S = rq.Cell("Rb85", [g, e1])
print(Rb_Cell_6S.couplings.nodes(data="gamma_lifetime"))
print(Rb_Cell_6S.couplings.edges(data="gamma_transition"))
print(Rb_Cell_6S.decoherence_matrix())
```

```
[((n=5, l=0, j=0.5), 0.0), ((n=6, l=1, j=1.5), 8.464336938490163)]
[((n=6, l=1, j=1.5), (n=5, l=0, j=0.5), 1.9967623792713765)]
[[0.         0.        ]
 [8.46433694 0.        ]]
```

You may notice here that `gamma_transition` value on the edge is less than the state lifetime, but the decoherence matrix still accounts for the entire state lifetime over all paths. What gives? Where did the rest of that decay rate come from if the `decoherence_matrix()` method works exactly the same in `Sensor` as it does in `Cell`? To answer that question, let us again inspect the graph edges completely.

```
print(Rb_Cell_6S.couplings.edges(data=True))
```

```
[((n=6, l=1, j=1.5), (n=5, l=0, j=0.5), {'gamma_transition': 1.9967623792713765,
↪'label': '((6, 1, 1.5),(5, 0, 0.5))', 'gamma_mismatch': 6.467574559218787})]
```

Aha! `rydiqule` has created an extra attribute on the edge called `gamma_mismatch` which pretty much does what it sounds like. It is added by `rydiqule` if the decay rates out of a particular state do not sum to the state lifetime. Since all terms starting with `"gamma"` on an edge are summed to calculate that particular term in the decoherence matrix, we maintain a transparent convention that preserves the functionality of `Sensor.decoherence_matrix` without any opaque magic. In fact, if you dig into the source code of `Cell`, you will not actually find a `Cell.decoherence_matrix`, it inherits the function directly from `Sensor`.

Default behavior

To demonstrate this behavior more completely, lets recreate this `Cell` with *all* possible decay paths out of $6P^{3/2}$ and see that there is then no mismatch to account for, all decays sum to the decay rate associated with the lifetime of the state. In this case, there are 4 allowed decays based on selection rules: $5S^{1/2}$ (the ground state of the atom), $6S^{1/2}$, $4D^{3/2}$, and $4D^{5/2}$

```
atom = "Rb85"
g = rq.ground_state(atom)

e_max = A_QState(6, 1, 1.5)

e1 = A_QState(6,0,0.5)
e2 = A_QState(4, 2, 1.5)
e3 = A_QState(4, 2, 2.5)
```

(continues on next page)

(continued from previous page)

```
Rb_Cell_6S_full_decay = rq.Cell(atom, [g, e1, e2, e3, e_max])

for s1, s2, gamma_mismatch in Rb_Cell_6S_full_decay.couplings.edges(data="gamma_
↪mismatch"):
    print(f"({s1},{s2}: {gamma_mismatch})")
```

```
((6, 0, 0.5),(5, 0, 0.5): 21.750359563468862)
((4, 2, 1.5),(5, 0, 0.5): 11.481506041906623)
((4, 2, 2.5),(5, 0, 0.5): 10.675163106446323)
((6, 1, 1.5),(5, 0, 0.5): None)
((6, 1, 1.5),(6, 0, 0.5): None)
((6, 1, 1.5),(4, 2, 1.5): None)
((6, 1, 1.5),(4, 2, 2.5): None)
```

We have `gamma_mismatch` on new edges (those out of states that ought to decay to $5P$ but don't because it is not in the system), but importatntly for this demonstration, there are no `gamma_mismatch` values associated with $6P^{3/2}$, because the total decay out of that state equals its `gamma_lifetime` value. We can check this to be sure in the following way:

```
gamma_lifetime = Rb_Cell_6S_full_decay.couplings.nodes[e_max]["gamma_lifetime"]
print(f"gamma_lifetime: {gamma_lifetime}")

total_decay = 0
for state in [g, e1, e2, e3]:
    total_decay += Rb_Cell_6S_full_decay.couplings.edges[e_max, state]["gamma_
↪transition"]
print(f"total gamma_transition: {total_decay}")
```

```
gamma_lifetime: 8.464336938490163
total gamma_transition: 8.464336938490163
```

You may notice in the above examples that `gamma_mismatch` is always sent to ground automatically. It is often suffient to assume this since in many systems population will eventually end up back in the ground state regardless of what path it took. Still this is not always the case, so how de we control this behavior? `rydiqule` has a couple of options that we control using the `gamma_mismatch` option in the constructor:

1. `"ground"`, the option demonstrated above, is the default. It adds a decoherent coupling from each state to $nS^{1/2}$, where n is the principle quantum number of the atom specified in the cell. If more that one sublevel exists for this state in the `Cell`, the decoherence will be divided equally amongst all sublevels. This coupling will be added regardless of whether the transition is dipole-allowed, since it is based on the assumption that all population will *eventually* decay to ground through some pathway. This option should be avoided for systems containing dark states.
2. `"all"` is an option that divides the mismatching decay values between all other calculated decay paths. The fraction of `gamma_mismatch` added to each edge is weighted by the existing calculated natural transition rates. So if state $|3\rangle$ has dipole-allowed decays to states $|2\rangle$ and $|1\rangle$ of 6 and 4 respectively, and a total `gamma_lifetime` in state $|3\rangle$ of 15, a `gamma_mismatch` of 3 and 2 will be added to the `(3,2)` and `(3,1)` edges respectively. Note that there must be a dipole-allowed decay for every state in the `Cell` other than the ground state or else selecting this option will error.
3. `"none"` does exactly what it sounds like: it will not add a single decay to the system for the computed state lifetime and decay rate mismatch. In this case, it is typically assumed that you will add a decay manually, but this leaves acounting for decay up to you.

In the future, `rydiqule` might support other options for which there is a compelling use case, but for now, anything besides the `ground` and `all` as described above will require manual specification of decays using the `none` option.

4.3.3 3.3 Transit broadening

As one final note on decoherent transitions, `Cell` treats transit broadening exactly the same is `Sensor`, although it does compute a transit broadening rate automatically based on atomic temperature, beam area, and atomic mass (beam is presumbed to be gaussian). So while the value of `transit_broadening` is computed automatically and stored as an attribute, you need to properly define the ensemble temperature and the optical beam area.

4.4 3. Couplings in `Cell`

Just like most things in `Cell`, couplings work much the same way as they do in `Sensor` but with some additional functionality.

4.4.1 3.1 Automatic Quantities

To see the basics of what is added automatically, let us create a simple 2-state system, add a single coupling between them, then inspect the resutling graph.

```
atom = "Rb85"
[g, e] = rq.D1_states(atom)

Rb_Cell_basic = rq.Cell(atom, [g, e])
Rb_Cell_basic.add_coupling((g, e), rabi_frequency=1, detuning=1, label="laser")

print(Rb_Cell_basic.couplings.edges(data=True))
```

```
[((n=5, l=0, j=0.5), (n=5, l=1, j=0.5), {'rabi_frequency': 1, 'detuning': 1, 'phase
↪': 0, 'kvec': (0, 0, 0), 'label': 'laser', 'coherent_cc': 1, 'dipole_moment': 1.
↪7277475900721146, 'q': 0}), ((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), {'gamma_
↪transition': 36.11450417508357, 'label': '((5, 1, 0.5),(5, 0, 0.5))'})]
```

Here we can see a coupling things. Firstly, the `dipole_moment` is calculated automatically. While this is not used directly in this case, it can come in handy for time-dependant couplings, and it can be a useful reference for other calculations you may want to do afterwards. Futhermore, if we add a coupling not in the rotating wave approximation, the `transition_frequency` will be calculated automatically. It is worth noting, however, that for very large transition frequencies, it can take a very long time (often prohibitively long) to solve in the time domain. `rydiqule` will warn you if you add a coupling without the RWA if the transition frequency is very high. If you are aware of this fact and want to suppress the warning, you can suppressed with `warnings.simplefilter('ignore', rq.RWAWarning)`, but it is often not desired to compute in the time domain when transition frequencies are quite large.

```
import warnings
warnings.simplefilter("ignore", rq.RWAWarning)

atom = "Rb85"
[g, e] = rq.D1_states(atom)

Rb_Cell_time = rq.Cell(atom, [g, e])
Rb_Cell_time.add_coupling((g, e), rabi_frequency=1, label="laser")

print(Rb_Cell_time.couplings.edges(data=True))
```

```
[((n=5, l=0, j=0.5), (n=5, l=1, j=0.5), {'rabi_frequency': 1, 'transition_frequency
↪': 2369435883.8824973, 'phase': 0, 'kvec': (0, 0, 0), 'label': 'laser',
↪'coherent_cc': 1, 'dipole_moment': 1.7277475900721146, 'q': 0}), ((n=5, l=1, j=0.
↪5), (n=5, l=0, j=0.5), {'gamma_transition': 36.11450417508357, 'label': '((5, 1,␣
↪0.5),(5, 0, 0.5))'})]
```

4.4.2 3.2 Aliasing manifolds with variables

While manifolds can be all defined manually, it is worth a brief sidebar here to discuss how to efficiently do it in `Cell` when manifolds can get quite large. We can already see one useful trick used in the above cell, by declaring `[g, e] = rq.D1_states(atom)`. In python, lists can be converted to individual single variables something like `[a,b,c] = my_list` as long as `my_list` has the number of elements that are upacked in the list defined.

Now we note that helper functions like `D1_states` and `expand_qnums`, which we introduce below, return lists of states. This allows for easy aliasing of states to the extent that we would like to apply them in couplings. We already showed this functionality to in the example above with `[g, e] = rq.D1_states(atom)`, but we can show some more involved examples here. Note that just like the constructor of `Cell`, states are passed to `expand_qnums` as a list.

While contrived, this example will show the utility of this approach well. Here we add a coupling from the ground state to the excited manifold, but add decoherences only from the $m_j = \pm 0.5$ states back to ground.

```
atom = "Rb85"
g = rq.ground_state(atom, splitting="fs")
e_fs = rq.D2_excited(atom, splitting="fs") #4 states total
print(f"states: {g, e_fs}\n")
[e1, e2, e3, e4] = rq.expand_qnums([e_fs])
print(f"m_j = +\-0.5 states: {e2, e3}\n")

my_cell = rq.Cell(atom, [g, e_fs])
my_cell.add_coupling((g,e1), rabi_frequency=1, detuning=0, label="laser")
my_cell.add_decoherence((e2, g), 1, label="foo")
my_cell.add_decoherence((e3, g), 1, label="bar")

print("couplings")
for edge in my_cell.couplings.edges(data=True):
    print(edge)

print(f"gamma matrix: \n {my_cell.decoherence_matrix()}")
```

```
states: ((n=5, l=0, j=0.5, m_j='all'), (n=5, l=1, j=1.5, m_j='all'))

m_j = +\-0.5 states: ((n=5, l=1, j=1.5, m_j=-0.5), (n=5, l=1, j=1.5, m_j=0.5))

couplings
((n=5, l=1, j=1.5, m_j=-1.5), (n=5, l=0, j=0.5, m_j=-0.5), {'gamma_transition': 38.
↪11316014416465, 'label': '((5, 1, 1.5, m_j=-1.5),(5, 0, 0.5, m_j=-0.5))'})
((n=5, l=1, j=1.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=-0.5), {'gamma_transition': 25.
↪408773429443098, 'label': '((5, 1, 1.5, m_j=-0.5),(5, 0, 0.5, m_j=-0.5))',
↪'gamma_foo': 1.0})
((n=5, l=1, j=1.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), {'gamma_transition': 12.
↪704386714721549, 'label': '((5, 1, 1.5, m_j=-0.5),(5, 0, 0.5, m_j=0.5))', 'gamma_
↪foo': 1.0})
((n=5, l=1, j=1.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=-0.5), {'gamma_transition': 12.
↪704386714721549, 'label': '((5, 1, 1.5, m_j=0.5),(5, 0, 0.5, m_j=-0.5))', 'gamma_
↪bar': 1.0})
((n=5, l=1, j=1.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=0.5), {'gamma_transition': 25.
↪408773429443098, 'label': '((5, 1, 1.5, m_j=0.5),(5, 0, 0.5, m_j=0.5))', 'gamma_
↪bar': 1.0})
((n=5, l=1, j=1.5, m_j=1.5), (n=5, l=0, j=0.5, m_j=0.5), {'gamma_transition': 38.
↪11316014416465, 'label': '((5, 1, 1.5, m_j=1.5),(5, 0, 0.5, m_j=0.5))'})
gamma matrix:
 [[ 0.          0.          0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.          0.          0.        ]
 [38.11316014  0.          0.          0.          0.          0.        ]
```

(continues on next page)

(continued from previous page)

```
 [26.40877343 13.70438671  0.          0.          0.          0.         ]
 [13.70438671 26.40877343  0.          0.          0.          0.         ]
 [ 0.          38.11316014  0.          0.          0.          0.         ]]
```

Breaking states out like this is not always necessary, but it can be useful in certain cases where you would like to only couple certain sublevels. Rather that retyping out `A_QState(n,l,j,...)` every time you reference it, you can assign it to a single terse variable without actually typing it out manually at all.

Often you only care about couplings between entire manifolds anyway, but this is a useful trick to have in your pocket.

4.4.3 3.3 Alternate specifications of `rabi_frequency`

Since `Cell` tries to support an interface to atomic physics that is a more close to real-life experiments, it has a couple of ways to speciy the power of a field beyond the `rabi_frequency` that is used in `Sensor`. As we have already demonstrated, you can define the rabi frequency of a coupling just fine (a `Cell` is just a `Sensor` after all), but `Cell` does also introduce a couple more options. All totalled, the three options for `rabi_frequency` specification are as follows:

1. Base specification of `rabi_frequency` just as in `Sensor`.
2. Specification of electric field, in V/m via the the `e_field` arguments. `rydiqule` will calculate the rabi_frequency based on the computed dipole moment.
3. Specification of both beam power, in watts and $\frac{1}{e^2}$ beam waist (radius) in meters via the `beam_power` and `beam_waist` arguments. In all cases, the arguments are provided as optional keyword arguments to the `add_coupling` function. These options are mutually exclusive. Also, none of this information will be stored on the graph directly - it will be converted to `rabi_frequency`, which is the only such quantity that is stored on the graph. We briefly demonstrate all of this below by defining a simple 3-level vee scheme.

```
atom = "Rb85"
g = rq.ground_state(atom)
D1_e = rq.D1_excited(atom)
D2_e = rq.D2_excited(atom)

RbCell_multi_rabi = rq.Cell(atom, [g, D1_e, D2_e])
RbCell_multi_rabi.add_coupling((g, D1_e), detuning=1, e_field=1)
RbCell_multi_rabi.add_coupling((g, D2_e), detuning=1, beam_power=1, beam_waist=0.
↪01)

#rabi_frequency is on all edges despite not being directly specified
print(RbCell_multi_rabi.couplings.edges.data("rabi_frequency"))
print(RbCell_multi_rabi.couplings.edges.data("e_field"))
print(RbCell_multi_rabi.couplings.edges.data("beam_power"))
```

```
[((n=5, l=0, j=0.5), (n=5, l=1, j=0.5), 0.13890429081447606), ((n=5, l=0, j=0.5),␣
↪(n=5, l=1, j=1.5), 429.7419875788027), ((n=5, l=1, j=0.5), (n=5, l=0, j=0.5),␣
↪None), ((n=5, l=1, j=1.5), (n=5, l=0, j=0.5), None)]
[((n=5, l=0, j=0.5), (n=5, l=1, j=0.5), None), ((n=5, l=0, j=0.5), (n=5, l=1, j=1.
↪5), None), ((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), None), ((n=5, l=1, j=1.5),␣
↪(n=5, l=0, j=0.5), None)]
[((n=5, l=0, j=0.5), (n=5, l=1, j=0.5), None), ((n=5, l=0, j=0.5), (n=5, l=1, j=1.
↪5), None), ((n=5, l=1, j=0.5), (n=5, l=0, j=0.5), None), ((n=5, l=1, j=1.5),␣
↪(n=5, l=0, j=0.5), None)]
```

4.4.4 3.4 Clebsch-Gordon coefficients

When defining couplings over manifolds, it is important to consider the Clebsch-Gordon coefficients (which we will refer to as CGC), which define the relating weighting of dipole moments between different sublevels. Recall that in `Sensor`, this is handled using the `coherent_cc` optional keyword argument in `add_coupling`. In `Cell`, the `coherent_cc` is filled in automatically using ARC's functions for the spherical component of the dipole matrix element. This section is really only intended to demonstrate that `coherent_cc` is calculated automatically in cell. For a more detailed description of how `rydiqule` handles coupling between manifolds of states, check the physics documentation. Here we just create a simple `Cell` on the D1 transition with fine structure splitting, and show that `coherent_cc` is populated on the graph even if we don't specify it explicitly.

```
atom = "Rb85"
g = rq.ground_state(atom, splitting="fs")
e = rq.D1_excited(atom, splitting="fs")
[g1, g2] = rq.expand_qnums([g])
[e1, e2] = rq.expand_qnums([e])

RbCell_cc = rq.Cell(atom, [g,e])
RbCell_cc.add_coupling((g,e), rabi_frequency=1, detuning=1, label="D1")
print(RbCell_cc.couplings.edges.data("coherent_cc"))
```

```
[((n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=1, j=0.5, m_j=-0.5), -0.816496580927726),␣
↪((n=5, l=0, j=0.5, m_j=0.5), (n=5, l=1, j=0.5, m_j=0.5), 0.816496580927726),␣
↪((n=5, l=1, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=-0.5), None), ((n=5, l=1,␣
↪j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), None), ((n=5, l=1, j=0.5, m_j=0.5),
↪ (n=5, l=0, j=0.5, m_j=-0.5), None), ((n=5, l=1, j=0.5, m_j=0.5), (n=5, l=0, j=0.
↪5, m_j=0.5), None)]
```

It is worth noting that `rydiqule` is making a very specific choice here regarding its convention regarding coupling coefficients. We choose to use the spherical dipole moment, since it is at least proportial to the Clebsch-Gordon coefficients and should ultimately produce the correct rabi frequency in the Hamiltonian. For consistency, we do not allow for manual specification of coupling coefficients in `Cell`, so make sure you are familiarized with the documentation above to ensure that the numbers in your simulation are correct.

4.4.5 3.5 Doppler Shifts

Recall that when we want to define a system with doppler broadening in a `Sensor`, we define the `vP` parameter either in the constructor or by accessing the attribute directly after construction, and then specifying the `kvec` parameter in couplings. In a `Cell`, however, the magnitude of the `kvec` parameter is calculated from the transition frequency by default, and the most probable speed is already set by the temperature. So when adding a coupling with doppler broadening in a `Cell`, we set only the newly-introduced `kunit` when couplings are defined. This is a tuple representing an `(x, y, z)` unit vector representing the direction of the plane wave of the field. Other than that, solving with doppler broadening in a `Cell` is identical to doing so in `Sensor`, and we can otherwise do everything the same way.

Note that for consistency with `Sensor`, the same `kvec` attribute is added to the graph edge, it is just computed automatically from `kunit`.

```
atom = "Rb85"
g = rq.ground_state(atom, splitting="fs")
e = rq.D1_excited(atom, splitting="fs")
[g1, g2] = rq.expand_qnums([g])
[e1, e2] = rq.expand_qnums([e])

RbCell_dop = rq.Cell(atom, [g,e])
RbCell_dop.add_coupling((g,e), rabi_frequency=1, detuning=1, label="D1", kunit=(1,
↪0,0))
print(RbCell_dop)

print(rq.solve_steady_state(RbCell_dop, doppler=True).rho)
```

```
<class 'rydiqule.cell.Cell'> object with 4 states and 2 coherent couplings.
States: [(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), (n=5, l=1, j=0.5,
↪ m_j=-0.5), (n=5, l=1, j=0.5, m_j=0.5)]
Coherent Couplings:
    ((5, 0, 0.5, m_j=-0.5),(5, 1, 0.5, m_j=-0.5)): {rabi_frequency: 1, detuning: 1,
↪ phase: 0, kvec: <parameter with 3 values>, label: D1_0, coherent_cc: -0.
↪816496580927726, dipole_moment: -1.7277475900721146, q: 0}
    ((5, 0, 0.5, m_j=0.5),(5, 1, 0.5, m_j=0.5)): {rabi_frequency: 1, detuning: 1,␣
↪phase: 0, kvec: <parameter with 3 values>, label: D1_1, coherent_cc: 0.
↪816496580927726, dipole_moment: 1.7277475900721146, q: 0}
Decoherent Couplings:
    ((5, 1, 0.5, m_j=-0.5),(5, 0, 0.5, m_j=-0.5)): {gamma_transition: 12.
↪03816805836119}
    ((5, 1, 0.5, m_j=-0.5),(5, 0, 0.5, m_j=0.5)): {gamma_transition: 24.
↪07633611672238}
    ((5, 1, 0.5, m_j=0.5),(5, 0, 0.5, m_j=-0.5)): {gamma_transition: 24.
↪07633611672238}
    ((5, 1, 0.5, m_j=0.5),(5, 0, 0.5, m_j=0.5)): {gamma_transition: 12.
↪03816805836119}
Energy Shifts:
    None
[ 0.00000000e+00 -1.09387364e-07  0.00000000e+00  0.00000000e+00
  4.97771560e-01  0.00000000e+00  1.09387364e-07 -1.86770118e-04
  0.00000000e+00  4.22260158e-06  0.00000000e+00  0.00000000e+00
  1.86770118e-04  0.00000000e+00  4.22260158e-06]
```

While using the atomic transition frequency to calculate the magnitude of the k-vector works well in most cases, it leads to inaccuracy in Doppler-averaged calculations if the field detuning is large relative to typical Doppler shifts. In this case, you can use the `kmag_detuning_correction` argument for a coupling to provide the average detuning relative to the atomic transition to be used when calculating the k-vector magnitude.

Note that passing this argument does not change the `detuning` argument, so this average shift should be accounted for in both places (ie `detuning = laser_scan + offsetFreq` and `kmag_detuning_correction=offsetFreq`).

4.5 4. Solving systems defined in `Cell`

Fortunately, this will be a straightforward section. Since `Cell` inherits `Sensor`, the mechanics of solving are identical between the two classes. There are a couple of additional considerations about automatically calculated quantities that we will go over in this section.

4.5.1 4.1 Just like `Sensor`!

Just like in `Sensor`, calling `rq.solve_steady_state` or `rq.solve_time` will produce a solution object containing information pertinent to the solve, as we demonstrate below.

```
atom = "Rb85"
g = rq.ground_state(atom)
e1 = rq.D2_excited(atom)
e2 = A_QState(6, 2, 2.5)

det = np.linspace(-1,1,11)
rabi = np.linspace(0.1, 1, 10)

RbCell_time = rq.Cell(atom, [g, e1, e2])
RbCell_time.add_coupling((g,e1), rabi_frequency=1, detuning=det, label="D1")
RbCell_time.add_coupling((e1, e2), rabi_frequency=rabi, detuning=1, label="upper")
```

(continues on next page)

(continued from previous page)

```
sol = rq.solve_steady_state(RbCell_time)
print(sol.rho.shape)
print(sol.axis_labels)
```

```
(11, 10, 8)
['D1_detuning', 'upper_rabi_frequency', 'density_matrix']
```

4.5.2 4.2 Observables in Cell

There are a couple other additions of note in `Cell` that are not present in `Sensor`. Specifically, some automatic calculations that, in `Sensor`, would be specified manually. Recall that the `kappa` and `eta` quantities, when relevant for calculating observables, would need to be spcecified as defined in the API docs. Since all the relevant parts are already in `Cell`, it cannot be specified in `Cell`, it must be computed automatically.

```
[g,e] = rq.D1_states(5)
RbCell_kappa_eta = rq.Cell("Rb85", [g, e])
RbCell_kappa_eta.add_coupling((g,e), rabi_frequency=1, detuning=0)

print(f"kappa: {RbCell_kappa_eta.kappa}")
print(f"eta: {RbCell_kappa_eta.eta}")
```

```
kappa: 14251202077.430082
eta: 0.9529640302753843
```

CHAPTER

FIVE

RYDIQULE PERFORMANCE

The primary purpose of this notebook is to provide a sense of how solve times in rydiqule scale with problem size. It also provides a practical example of the advantages of using rydiqule for large numbers of semi-classical master-equation solving relative to other tools.

```
import rydiqule as rq
import numpy as np
import matplotlib.pyplot as plt
import time
```

5.1 Steady-state solve time scaling versus basis size

We begin by showing how the solve time for a single set of Equations of Motion scales with the number of states in the basis b.

For this benchmark, we randomize each problem to be solved, ensuring that there are around $b * \log(b)$ couplings with a random coupling strength, and each state decays to the ground state with a random rate. All detunings are held to be 0.

```
def random_ss_sensor(n_states, n_couplings, rabi_bounds=[0.1,10], detuning_
↪bounds=[0,0], decoherence_bounds=[0,0.5]):
    if type(n_states) is not int:
        n_states = n_states.item()
    s = rq.Sensor(n_states)

    decoherences = np.random.uniform(decoherence_bounds[0], decoherence_bounds[1],␣
↪size=n_states)
    gamma = np.zeros((n_states, n_states))
    gamma[:,0] = decoherences
    s.set_gamma_matrix(gamma)

    possible_coups = np.array(np.triu_indices(n_states, 1), dtype=int).T
    idx = np.random.choice(np.arange(len(possible_coups)), n_couplings,␣
↪replace=False)
    coups = possible_coups[idx]

    rabis = np.random.uniform(rabi_bounds[0], rabi_bounds[1], size=n_couplings)
    detunings = np.random.uniform(detuning_bounds[0], detuning_bounds[1], size=n_
↪couplings)

    lasers = []
    for i in range(n_couplings):
        lasers.append({"states": (int(coups[i][0]), int(coups[i][1])), "rabi_
↪frequency": rabis[i], "detuning": detunings[i]})

```

(continues on next page)

(continued from previous page)

```
    s.add_couplings(*lasers)

    return s
```

To provide some statistical measure without making the notebook run too long in a single sitting, we average 4 runs per point.

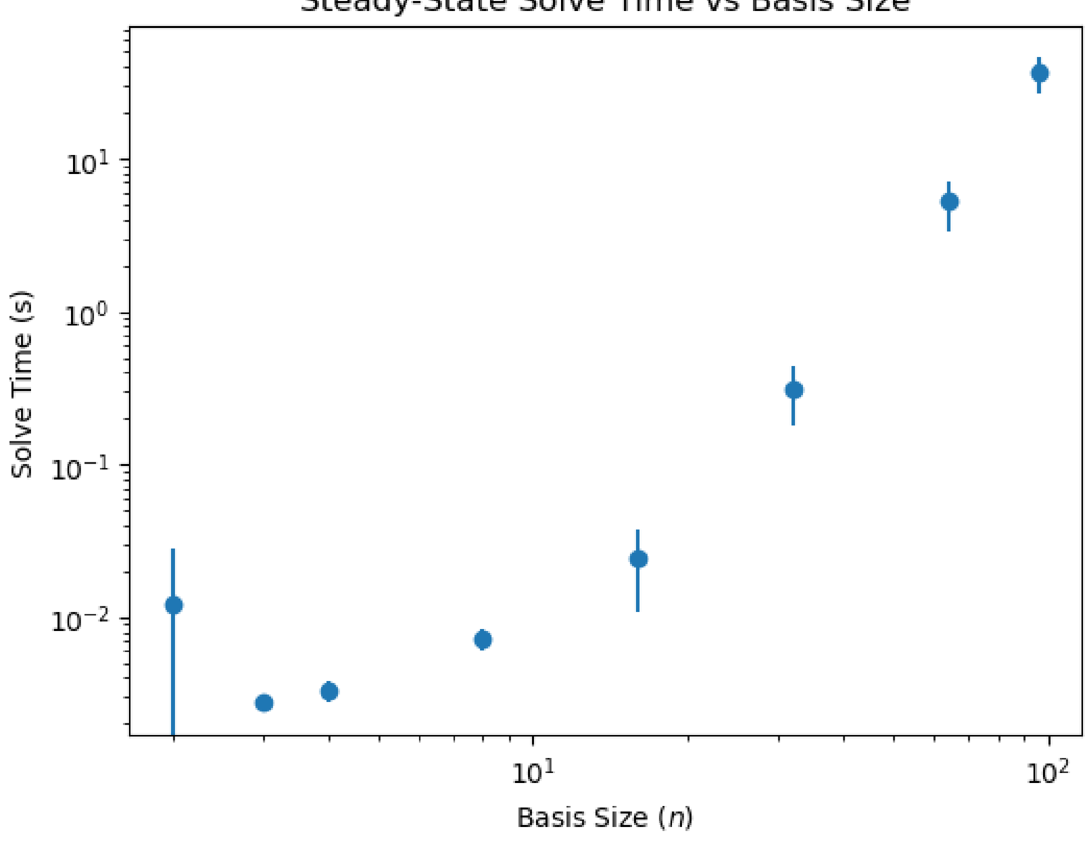
```
n_sims = 4

n_states = np.array([2, 3, 4, 8, 16, 32, 64, 96])
n_couplings = np.array(n_states*np.log(n_states), dtype=int)
print(n_states)
print(n_couplings)
```

```
[ 2  3  4  8 16 32 64 96]
[  1   3   5  16  44 110 266 438]
```

```
times = np.zeros((n_sims, len(n_states)))

for j in range(len(n_states)):
    for i in range(n_sims):
        print(f'Solving {n_states[j]:d} states, round {i+1:d}/{n_sims:d}', end='\r
↪')
        s = random_ss_sensor(n_states[j], n_couplings[j])
        tic = time.perf_counter()
        _ = rq.solve_steady_state(s)
        toc = time.perf_counter()
        times[i,j] = (toc - tic)
```

```
Solving 96 states, round 4/4
```

```
fig, ax = plt.subplots(1)

ave_times = times.mean(axis=0)
std_times = times.std(axis=0)

ax.errorbar(n_states, ave_times, yerr=std_times, fmt='o')
ax.set_title('Steady-State Solve Time vs Basis Size')
ax.set_xlabel('Basis Size ($n$)')
ax.set_ylabel('Solve Time (s)')
ax.set_yscale('log')
ax.set_xscale('log')
```



5.2 Steady-state solve time scaling versus stack size

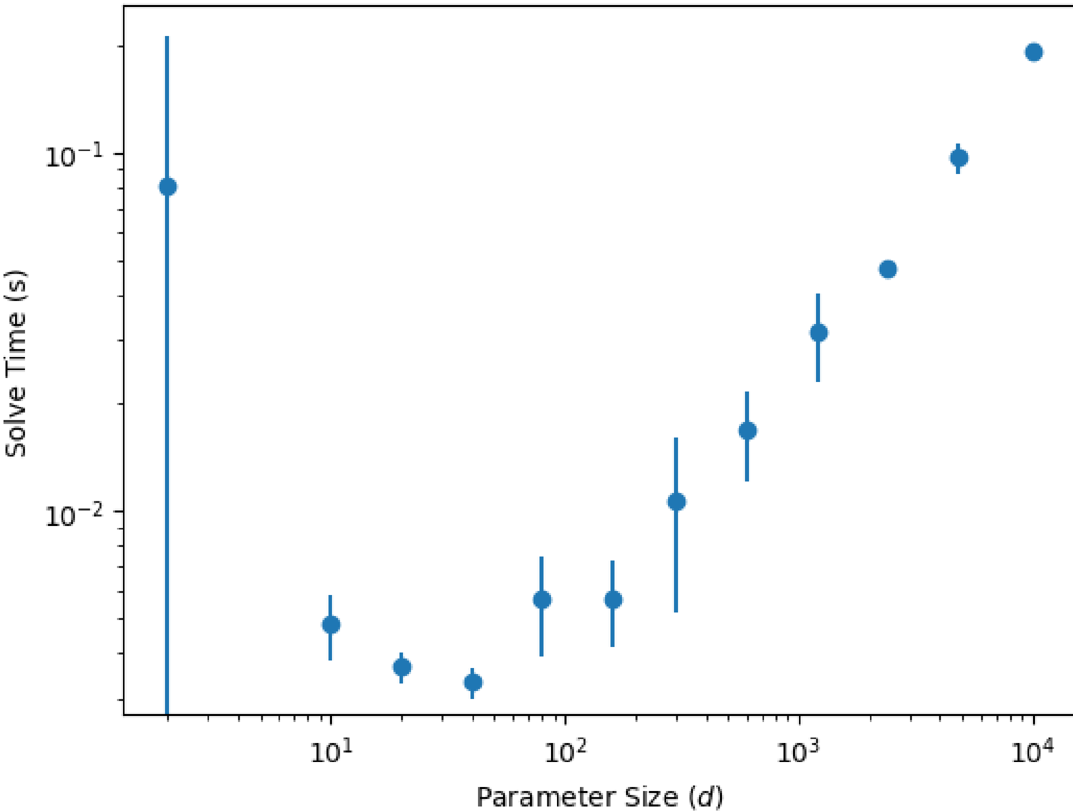
Here we see how solve time scales versus the number of Equations of Motion to be solved for a system with a basis size of 3. This is done by varying how many probe detuning points we sample.

```
couple = {'states': (1,2), 'rabi_frequency':2*np.pi*1, 'detuning':2*np.pi*0, 'label
↪':'couple'}

sts = rq.Sensor(3)
sts.add_couplings(couple)
sts.add_decoherence((1,0), 2*np.pi*6.0666)
sts.add_decoherence((2,1), 2*np.pi*0.1)
```

```
n_sims = 4
ssize = np.array([2, 10, 20, 40, 80, 160, 300, 600, 1200, 2400, 4800, 10000])
```

```
times_stack = np.zeros((len(ssize), n_sims), dtype=float)

for i, p_pts in enumerate(ssize):
    for j in range(n_sims):
        probe = {'states': (0,1), 'rabi_frequency':2*np.pi*0.1,
                 'detuning':2*np.pi*np.linspace(-100, 100, p_pts), 'label':'probe'}
        sts.add_couplings(probe)
        tic = time.perf_counter()
        _ = rq.solve_steady_state(sts)
        toc = time.perf_counter()
        times_stack[i,j] = (toc-tic)
```

```
fig, ax = plt.subplots(1)

ave_times = times_stack.mean(axis=1)
std_times = times_stack.std(axis=1)

ax.errorbar(ssize, ave_times, yerr=std_times, fmt='o')
ax.set_title('Steady-State Solve Time vs Parameter Size')
ax.set_xlabel('Parameter Size ($d$)')
ax.set_ylabel('Solve Time (s)')
ax.set_yscale('log')
ax.set_xscale('log')
```



5.3 Time-dependent solve times versus stack size

Here we repeat the test, but with a time-dependent coupling. Note that time-dependent solving is slower than steady-state by a fair bit. This is not aided by our insistence on allowing arbitrary python functions for the time dependence, which makes the back-end implementations tricky.

```
couple = {'states': (1,2), 'rabi_frequency':2*np.pi*1, 'detuning':2*np.pi*0, 'label
↪':'couple',
          'time_dependence': lambda t: np.sin(2*np.pi*2*t)}

tts = rq.Sensor(3)
tts.add_couplings(couple)
tts.add_decoherence((1,0), 2*np.pi*6.0666)
tts.add_decoherence((2,1), 2*np.pi*0.1)
```

```
n_sims = 4
tsize = np.array([2, 10, 20, 40, 80, 160, 320])
```

Rydiqule's standard time-solving backend is scipy's `solve_ivp`.

```
ttimes_stack = np.zeros((len(tsize), n_sims), dtype=float)

for i, p_pts in enumerate(tsize):
    for j in range(n_sims):
        print(f'Solving {p_pts} detunings, round {j+1}/{n_sims}', end='\r')
        probe = {'states': (0,1), 'rabi_frequency':2*np.pi*0.1,
                 'detuning':2*np.pi*np.linspace(-100, 100, p_pts), 'label':'probe'}
        tts.add_couplings(probe)
        tic = time.perf_counter()
        _ = rq.solve_time(tts, end_time=10, num_pts=100)
        toc = time.perf_counter()
        ttimes_stack[i,j] = (toc-tic)
```

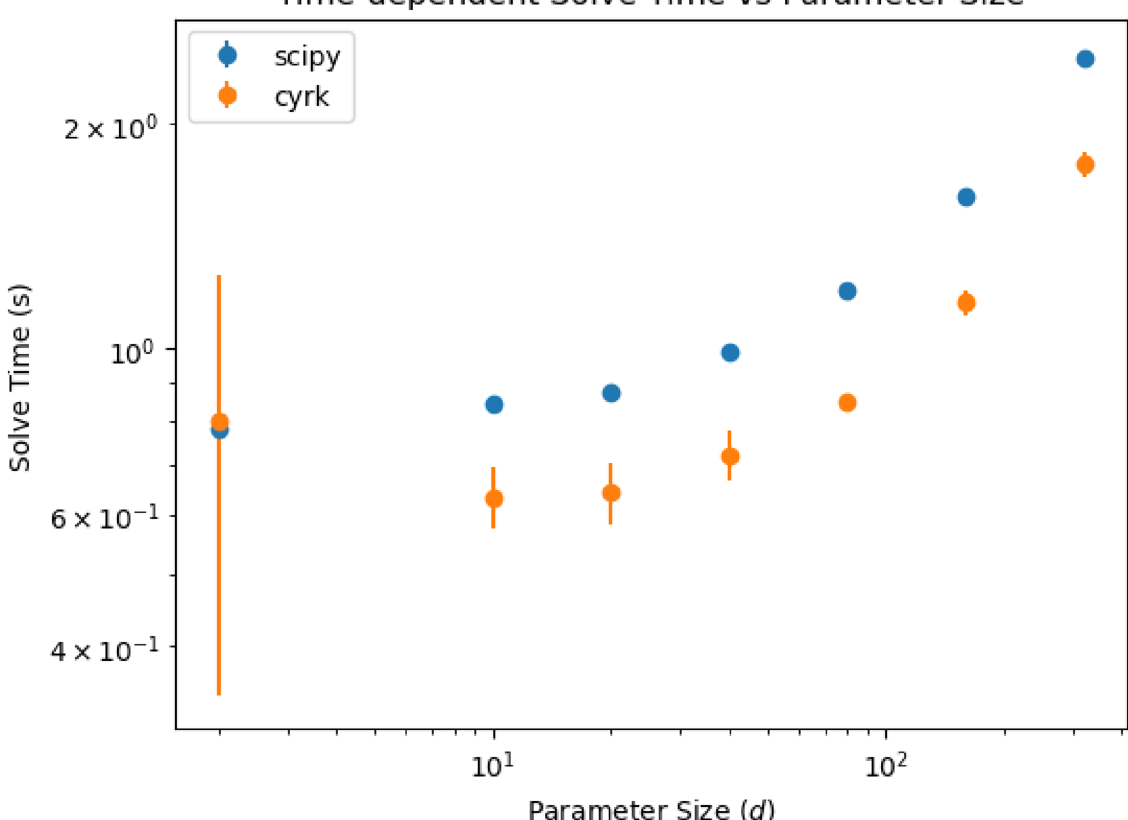
```
Solving 320 detunings, round 4/4
```

Rydiqule also has some support for a compiled version of `solve_ivp` provided by the `CyRK` package. It can be faster for some kinds of problems, but it is not compatible with all problems in rydiqule. For example, the optional `flat` argument used here provides the best performance, but is not compatible with doppler-broadened solves.

```
cttimes_stack = np.zeros((len(tsize), n_sims), dtype=float)

for i, p_pts in enumerate(tsize):
    for j in range(n_sims):
        print(f'Solving {p_pts} detunings, round {j+1}/{n_sims}', end='\r')
        probe = {'states': (0,1), 'rabi_frequency':2*np.pi*0.1,
                 'detuning':2*np.pi*np.linspace(-100, 100, p_pts), 'label':'probe'}
        tts.add_couplings(probe)
        tic = time.perf_counter()
        _ = rq.solve_time(tts, end_time=10, num_pts=100, solver='cyrk', eqns='flat
↪')
        toc = time.perf_counter()
        cttimes_stack[i,j] = (toc-tic)
```

```
Solving 320 detunings, round 4/4
```

```
fig, ax = plt.subplots(1)

ave_times = ttimes_stack.mean(axis=1)
std_times = ttimes_stack.std(axis=1)

ax.errorbar(tsize, ave_times, yerr=std_times, fmt='o', label='scipy')
ax.errorbar(tsize, cttimes_stack.mean(axis=1), yerr=cttimes_stack.std(axis=1), fmt=
↪'o', label='cyrk')
ax.legend()
ax.set_title('Time-dependent Solve Time vs Parameter Size')
ax.set_xlabel('Parameter Size ($d$)')
ax.set_ylabel('Solve Time (s)')
ax.set_yscale('log')
ax.set_xscale('log')
```



5.4 Comparison to qutip

While qutip primarily targets a different (and wider) set of problems, we can compare solving a simple 2-level system over a set of parameters. We note that qutip is not designed or advertised to be effective at solving semi-classical quantum systems with large numbers of atomic levels or iterated over many parameters. Rydiqule, on the other hand, is specifically designed to solve that subset of problems, as will be seen below.

First, we solve the demonstration system in rydiqule. It has 101 detunings and 10 rabi frequencies, and we want to plot the imaginary part of $\rho_{1,0}$.

```
detunings = np.linspace(-50, 50, 101)
rabis = np.geomspace(0.1, 25, 10)
gamma = 6.0666

probe = {'states': (0,1), 'rabi_frequency':2*np.pi*rabis, 'detuning':2*np.
↪pi*detunings, 'label':'probe'}

s2 = rq.Sensor(2)
s2.add_couplings(probe)
s2.add_decoherence((1,0), 2*np.pi*gamma)
```

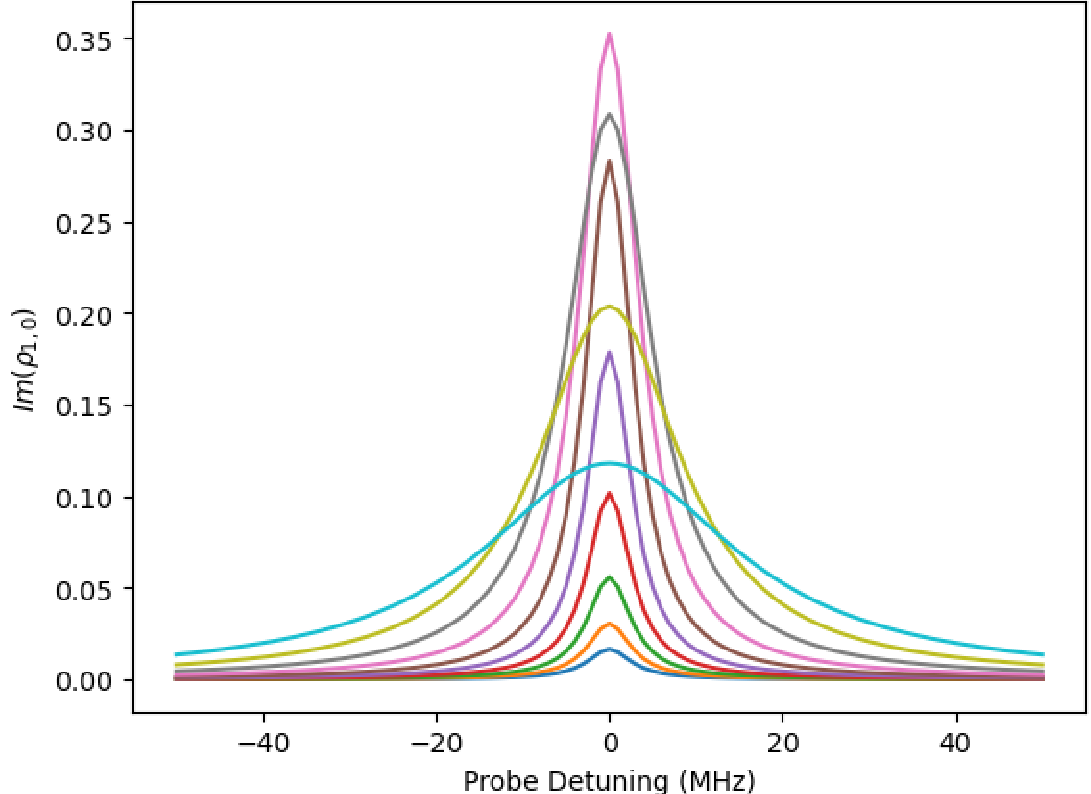
```
%%timeit
_ = rq.solve_steady_state(s2)
```

```
12.6 ms ± 908 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
sols_rq2 = rq.solve_steady_state(s2)
```

```
fig, ax = plt.subplots(1)

ax.plot(detunings, sols_rq2.rho_ij(1,0).imag)
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('$Im(\\rho_{1,0})$')
```

```
Text(0, 0.5, '$Im(\\rho_{1,0})$')
```



Constructing the same hamiltonian in qutip is done by expressing it as sums of Pauli matrices. Note that each parameter set requires re-building the Hamiltonian at each point, done via a nested python loop. It is difficult to scale the problem configuration to large numbers of atomic levels. And each set of parameters to be iterated over requires a separate python loop. The incurred interpreter overhead is substantial, giving a solve time more than 3 orders of magnitude slower.

```
import qutip
```

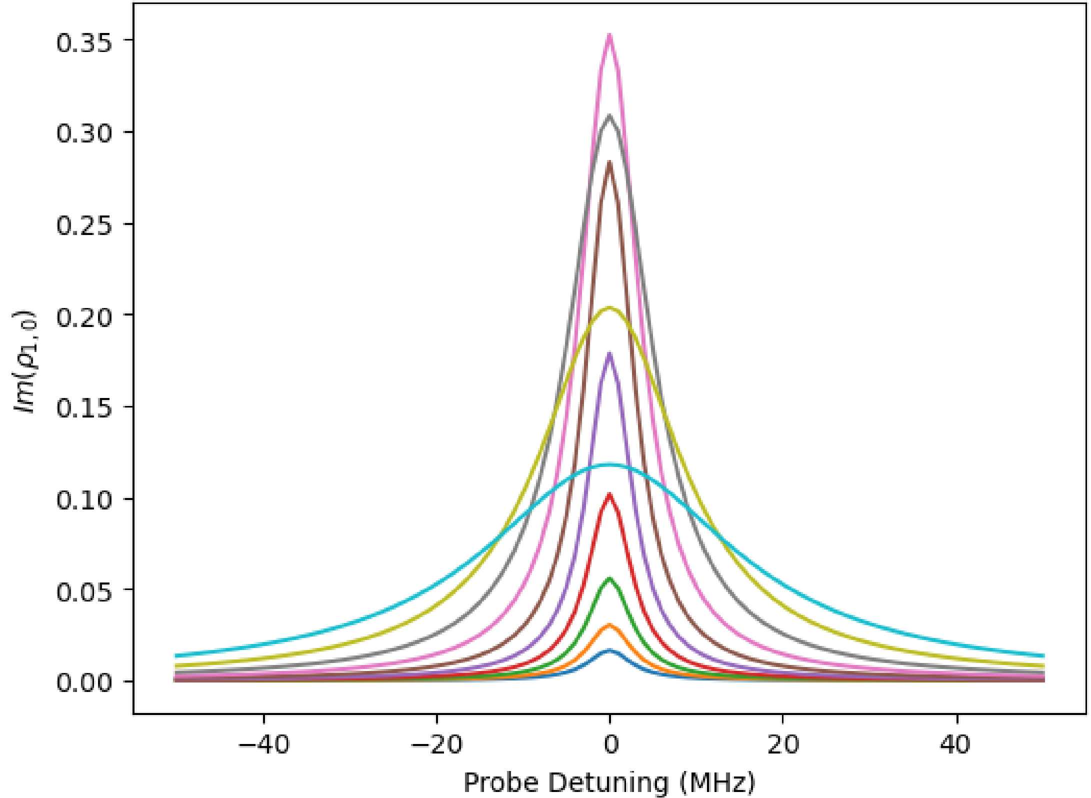
```
sm = qutip.destroy(2)
c_op_list = [np.sqrt(2*np.pi*gamma) * sm] # note: qutip wants root of the rate

psi0 = qutip.basis(2, 0)
psi1 = qutip.basis(2, 1)

def q_solve(det, rabi):

    H0 = (2*np.pi*det*qutip.sigmaz() + 2*np.pi*rabi*qutip.sigmax())/2

    rho_ss = qutip.steadystate(H0, c_op_list, method='direct', solver='solve').
↪full()

    return rho_ss
```

```
%%time
sols_qtp = np.empty((len(detunings), len(rabis), 2, 2), dtype = np.complex128)

for i, det in enumerate(detunings):
    for j, rabi in enumerate(rabis):
        sols_qtp[i,j] = q_solve(det, rabi)
```

```
CPU times: total: 20 s
Wall time: 20 s
```

```
fig, ax = plt.subplots(1)

ax.plot(detunings, sols_qtp[:,:,0,1].imag)
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('$Im(\\rho_{1,0})$')
```

```
Text(0, 0.5, '$Im(\\rho_{1,0})$')
```



5.4.1 Time-dependence comparison

We compare the same problem, but this time with a time-dependent Hamiltonian. Here QuTiP provides many highly-optimized time-integrators and is generally faster for a solve on a single set of parameters.

We start by solving the simple system in rydiqule for a single set of parameters.

```
gamma = 6.0666

probe = {'states': (0,1), 'rabi_frequency':2*np.pi*1, 'detuning':2*np.pi*0, 'label
↪':'probe',
         'time_dependence': lambda t: np.sin(2*np.pi*2*t)}
```

(continues on next page)

(continued from previous page)

```
s2t = rq.Sensor(2)
s2t.add_couplings(probe)
s2t.add_decoherence((1,0), 2*np.pi*gamma)

init_cond_complex = np.array(((1,0),
                              (0,0)), dtype=float)
init_cond = rq.sensor_utils.convert_complex_to_dm(init_cond_complex)
print(init_cond)
end_time = 10 # us
n_pts = 400
```

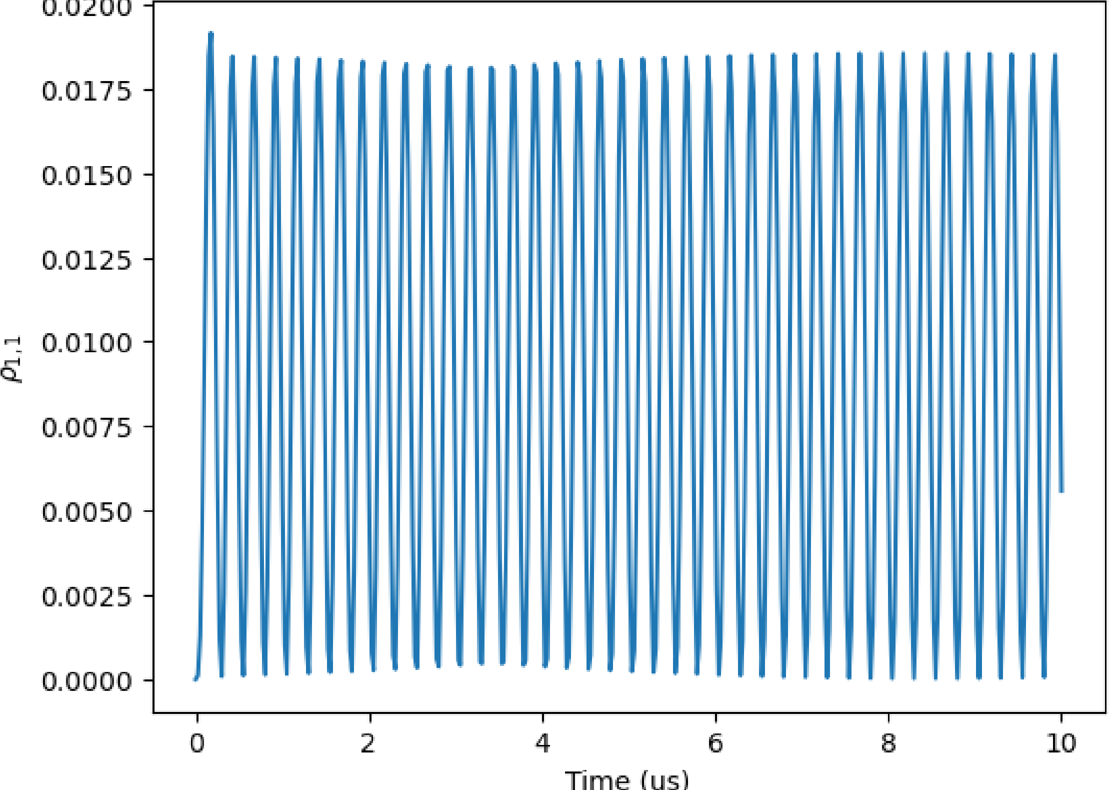
```
[0. 0. 0.]
```

```
%%time
sols_rq2t = rq.solve_time(s2t, end_time, n_pts, init_cond=init_cond)
```

```
CPU times: total: 93.8 ms
Wall time: 98.8 ms
```

```
fig, ax = plt.subplots(1)

ax.plot(sols_rq2t.t, sols_rq2t.rho_ij(1,1))
ax.set_xlabel('Time (us)')
ax.set_ylabel('$\\rho_{1,1}$')
```

```
Text(0, 0.5, '$\\rho_{1,1}$')
```



We solve the same problem in qutip, using its standard `mesolve` with the optional cython-compiled backend enabled.

```
sm = qutip.destroy(2)
c_op_list = [np.sqrt(2*np.pi*gamma) * sm]

psi0 = qutip.basis(2, 0)
psi1 = qutip.basis(2, 1)
w = 2

t_list = np.linspace(0, end_time, n_pts)

def q_tsolve(det, rabi):

    H0 = (2*np.pi*det*qutip.sigmaz())/2
    H1 = 2*np.pi*rabi*qutip.sigmax()/2
    args = {'w': w}

    H = [H0, [H1, "np.sin(2*np.pi*w*t)"]]

    rho_t = qutip.mesolve(H, psi0, t_list, c_op_list, args=args)

    return rho_t
```

Ensure a first run to allow first compilation to occur. Requires optional dependencies: `cython` and `filelock`.

```
sols_qtp2 = q_tsolve(0, 1.0)
```

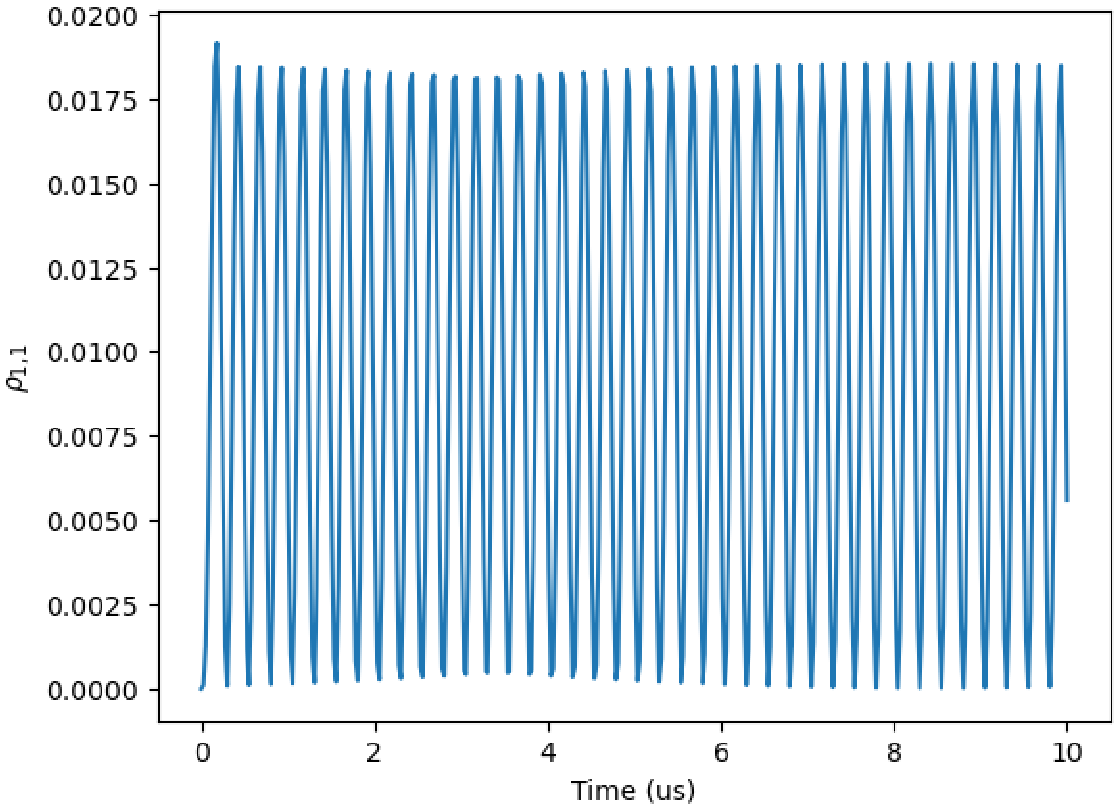
QuTiP's (mostly) standard solver is over x4 faster.

```
%%timeit
sols_qtp2 = q_tsolve(0, 1.0)
```

```
22.3 ms ± 1.07 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
fig, ax = plt.subplots(1)

prob_11 = qutip.expect(psi1 * psi1.dag(), sols_qtp2.states)

ax.plot(t_list, prob_11)
ax.set_xlabel('Time (us)')
ax.set_ylabel('$\\rho_{1,1}$')
```

```
Text(0, 0.5, '$\\rho_{1,1}$')
```



5.4.2 Time-dependent comparison over many parameters

If we want to solve the same time-dependent problem for many parameters, this is easy to set up in rydiqule but uses nested for loops for qutip (which incurs interpreted overhead similar to the steady-state example above).

```
detunings = np.linspace(-50, 50, 101)
rabis = np.geomspace(0.1, 25, 10)
gamma = 6.0666

probe = {'states': (0,1), 'rabi_frequency':2*np.pi*rabis, 'detuning':2*np.
↪pi*detunings, 'label':'probe',
         'time_dependence': lambda t: np.sin(2*np.pi*2*t)}

s2t = rq.Sensor(2)
s2t.add_couplings(probe)
s2t.add_decoherence((1,0), 2*np.pi*gamma)

init_cond = rq.solve_steady_state(s2t).rho
print(init_cond.shape)
end_time = 10 # us
n_pts = 400
```

```
(101, 10, 3)
```

```
%%time
sols_rq2t = rq.solve_time(s2t, end_time, n_pts, init_cond=init_cond)
```

```
CPU times: total: 1.56 s
Wall time: 1.56 s
```

To make pre-allocating the results easier, we use qutip's ability to automatically calculate expectation values for a desired parameter at solve time.

```
sm = qutip.destroy(2)
c_op_list = [np.sqrt(2*np.pi*gamma) * sm]

psi0 = qutip.basis(2, 0)
psi1 = qutip.basis(2, 1)
w = 2

t_list = np.linspace(0, end_time, n_pts)

def q_tsolve(det, rabi):

    H0 = (2*np.pi*det*qutip.sigmaz())/2
    H1 = 2*np.pi*rabi*qutip.sigmax()/2
    args = {'w': w}

    H = [H0, [H1, "np.sin(2*np.pi*w*t)"]]

    # calculate desired output here since pre-allocating and storing density␣
↪matrices is annoying
    rho11_t = qutip.mesolve(H, psi0, t_list, c_op_list, e_ops = [psi1*psi1.dag()],␣
↪args=args).expect[0]

    return rho11_t
```

We see that the interpreter overhead is significant, leading to over an order of magnitude slower total calculation time versus rydiqule.

```
%%time
sols_qtp2p = np.empty((len(detunings), len(rabis), n_pts), dtype = np.complex128)
size = len(detunings)*len(rabis)

for i, det in enumerate(detunings):
    for j, rabi in enumerate(rabis):
        print(f'Solving {(i)*len(rabis)+(j+1)}/{size}', end='\r')
        sols_qtp2p[i,j,:] = q_tsolve(det, rabi)
```

```
CPU times: total: 58.1 s
Wall time: 56.5 s
```

```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.1.1.dev37
Installation Path:    ~\src\rydiqule_public\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.0.1
SciPy Version:        1.16.0
Matplotlib Version:   3.10.5
ARC Version:          3.9.0
```

(continues on next page)

(continued from previous page)

```
Python Version:        3.12.3
Python Install Path:   ~\miniconda3\envs\qutip
Platform Info:         Windows (AMD64)
CPU Count and Freq:    16 @ 3.91 GHz
Total System Memory:   256 GB
```

```
qutip.about()
```

```
QuTiP: Quantum Toolbox in Python
================================
Copyright (c) QuTiP team 2011 and later.
Current admin team: Alexander Pitchford, Nathan Shammah, Shahnawaz Ahmed, Neill␣
↪Lambert, Eric Giguère, Boxi Li, Simon Cross, Asier Galicia, Paul Menczel, and␣
↪Patrick Hopf.
Board members: Daniel Burgarth, Robert Johansson, Anton F. Kockum, Franco Nori and␣
↪Will Zeng.
Original developers: R. J. Johansson & P. D. Nation.
Previous lead developers: Chris Granade & A. Grimsmo.
Currently developed through wide collaboration. See https://github.com/qutip for␣
↪details.

QuTiP Version:      5.2.1
Numpy Version:      2.0.1
Scipy Version:      1.16.0
Cython Version:     3.1.3
Matplotlib Version: 3.10.5
Python Version:     3.12.3
Number of CPUs:     16
BLAS Info:          Generic
INTEL MKL Ext:      None
Platform Info:      Windows (AMD64)
Installation path:  c:\Users\naqsL\miniconda3\envs\qutip\Lib\site-packages\qutip

Installed QuTiP family packages
-------------------------------

No QuTiP family packages installed.

================================================================================
Please cite QuTiP in your publication.
================================================================================
For your convenience a bibtex reference can be easily generated using `qutip.
↪cite()`
```

CHAPTER

SIX

CHANGELOG

6.1 v2.1.3

6.1.1 Improvements

- Add `kmag_detuning_correction` keyword argument to `Cell.add_single_coupling`. Default behavior is to only use the atomic transition frequency to calculate the k-vector magnitude for a coupling with `kunit` provided, but this leads to inaccurate doppler averages if the coupling detuning is large relative to Doppler shifts. This new argument allows passing in a detuning to be applied to the calculation of the k-vector magnitude for a coupling.

6.1.2 Bug Fixes

- Fix issue where `Solution.coupling_rabi` would fail for a coupling group that contains non-dipole-allowed transitions (as common in `Cell` with sublevel structure).
- Fix incorrect `Sensor.dm_basis` labels. E.g. `01_real`, `01_imag` is actually `10_real`, `10_imag`. For real terms, change is purely cosmetic, but imaginary terms there is a sign difference. This change brings `dm_basis` labels to agree with the actual basis definitions (ie `Solution.rho_ij`).

6.2 v2.1.2

6.2.1 Improvements

- Improved documentation based on JOSS reviewer @a-eghrari feedback. Biggest additions include more detailed contribution guidelines and a quantitative discussion of rydiqule performance (largely lifted from Comp Sci Comms article).
- Added the `Solution.complex_rho` property which provides ready conversion of the solved density matrices into the complex basis with ground state present.
- Made project `uv` compatible and added installation docs describing how to use it.
- Updated license metadata to follow PEP 639 standard
- Improve scannable parameter handling so that all sequences are saved to the graph as numpy arrays. Also ensures proper handling of length-1 and length-0 sequences.
- Modified the `about` function to provide numpy's BLAS/LAPACK backend information.
- Updates stale example notebooks to incorporate v2.1.1 bugfixes
- Improved documentation based on JOSS reviewer @nikolasibalic feedback.

6.2.2 Bug Fixes

- Fix `convert_complex_to_dm` to properly return arrays with real dtype.
- Fix `Cell.kappa` and `Cell.eta` to properly introspect `q` when first edge on the graph happens to not be dipole-allowed.

- Fix `draw_diagram` to handle complex time-dependent functions correctly.
- Fix `cyrk_solve` flat diffEq backend to properly handle EOM stacks in both steady-state and time-dependent problems.

6.2.3 Deprecations

6.3 v2.1.1

6.3.1 Improvements

- `leveldiagram` and `arc` dependencies are now lazy loaded at first use (`draw_diagram` and `Cell` instantiation, respectively). This improves `import rydiqule as rq` time by about a second and ensures second-order dependencies such as matplotlib and sympy are not imported unless explicitly needed.
- Expand Observables documentation to describe generalized paradigm in version 2, including how to calculate custom observables.

6.3.2 Bug Fixes

- Minor changes to the precision of hard-coded values in unit tests. ARC v3.9.0 updated the precision of a constant used in its internal calculations that shifts calculated dipole moment values by a small amount. It is expected that other plots and values will differ by small amounts when using `Cell` and ARC v3.9.0
- Fixed bug where `solve_doppler_analytic` ignored parameter time-dependence. It now follows the behavior of other steady-state solvers by constructing the Hamiltonian and equations of motion at `t=0`.
- Ensure that `Cell.eta`, `Cell.kappa`, and `Cell.probe_freq` properly cache their values as intended.
- Fix bug where `Cell` would not add transit broadening automatically, as intended.
- Fix errors in `gamma_mismatch` calculations when coupling groups are used. Also fix issue that prevented `gamma_mismatch='all'` from working if only 1 dephasing is present that needs modification.
- Update RF heterodyne example notebook to use new `Cell` syntax.
- Properly mark slow/high-memory analytic doppler test.

6.4 v2.1.0

6.4.1 Improvements

- Fully support 1D/2D/3D doppler solves using the analytic-enabled solver, `solve_doppler_analytic`. The analytic average is applied to a single dimension, with other present dimensions treated numerically. This provides notable decreases in computation times and memory footprints and increases in solution accuracy in all spatial dimensions.
- Added an example notebook, `Analytic Doppler Solver`, showcasing the new `solve_doppler_analytic`.
- Extended documentation and unit unit-testing to include hybrid solver implementation.

6.4.2 Deprecations

- `doppler_1d_exact` has been deprecated in favor of the more complete `solve_doppler_analytic` which handles 1D/2D/3D analytic-enabled averaging

6.5 v2.0.0

6.5.1 Improvements

- Added ability to use arbitrary tuples to define a state.

- Added concept of `StateSpec`, which is a tuple with a nested list for at least one element. Rydiqule will interpret this as a list of state tuples expanded along the nested element. For example, `(0, [-1, 0, 1])` is interpreted as the group of states `[(0, -1), (0, 0), (0, 1)]`.
- Added ability to couple groups of states to a single field using a single function. This involves several modifications:
  - The `Sensor.add_coupling_group` has been added which takes two lists of states and couples all combinations of states. Relative scaling of each sub-coupling is controlled via a `coupling_coefficients` dictionary. If an entry is missing from this dictionary, that sub-coupling will not be added.
  - The previous `Sensor.add_coupling` has has been renamed to `Sensor.add_single_coupling`.
  - The new `Sensor.add_coupling` function now takes either a pair of states or a pair of `StateSpec` (which define multiple states), and dispatches appropriately.
  - Identical functionality now exists for the `add_decoherence`, `add_self_broadening`, and `add_energy_shift` functions.
  - Added a string representation of `Sensor` so that a summary of the data on its graph is shown when calling `print(<sensor>)`.
- Complete overhaul of `Cell`.
  - States are now specified using the named tuple class `A_QState`, which tracks quantum numbers explicitly.
  - `Cell` now supports calculations with and without sublevels defined.
    * If sublevels are not defined (the NLJ basis), results now use an averaged dipole moment. Otherwise, calculations should be identical to `Cell` from v1. See *Migrating Cell from v1 to v2* for further details.
    * Sublevels can now be defined, in either the fine structure (FS) or hyperfine structure (HFS) bases. Couplings between bases are also supported. This support heavily relies on new coupling group functionality of `Sensor` described above.
  - A new interface to ARC calculations has been defined, `RQ_AlkaliAtom`, which defines methods for calculating atomic parameters between pairs of `A_QState`.
  - The `kvec` parameter for couplings is replaced with `kunit`, the unit propagation axis. Necessary prefactors for calculation are automatically applied using ARC functionality. See *Migrating Doppler averaging from v1 to v2* for details.
- Definition of `kvec` has been redefined to be the field k-vector, instead of the most probable velocity vector. The most probable speed `vP` must now be provided separately either via init keyword argument (i.e. `Sensor(4, vP=250)`) or by setting the `Sensor.vP` attribute. See *Migrating Doppler averaging from v1 to v2* for details.
- Allow the `about` function to optionally print numpy backend information.
- Created custom `RydiquleError` and `RydiquleWarning` classes and subclasses to denote errors and warnings specific to Rydiqule. By default, `RydiquleError` will suppress the raise statement in the traceback via patches to exception hooks. This can be disabled by calling `rq.set_debug_state(True)`. `rq.set_debug_state` also controls debugging print statements during Sensor/Cell creation.
- Added utility functions for converting density matrices between a standard complex basis and rydiqule's real with ground removed computational basis (`convert_dm_to_complex` and `convert_complex_to_dm`). Added utility to add ground state to real computational basis results `convert_to_full_dm`. Also added utility to confirm density matrices are physical (`check_positive_semi_definite`). This functionality is now used in `solve_time` to ensure user-provided `init_cond` are physical.
- Improved the ergonomics of `Sensor.zip_parameters` so that parameters to be zipped are specified by a dictionary keyed by coupling with entries that specify the parameter.
- Changed `Sensor.add_energy_shift` to to be more in line with couplings, so that it works as a dispatch function for single shifts or groups. Also added `add_energy_shift_group` which will add and zip a group of energy shifts.

- Package versioning is now handled by `setuptools_scm` which introspects the version based on git tags (if present). We also use this functionality to dynamically update the version on import when running an from an editable install, to account for local development.
- Overhaul of the observable functions of `sensor_solution` to use a more physical definition of observable defined by the trace of the density matrix times an operator. Additionally, those functions are transparent to allow more flexible definitions of observables.
- `sensor_solution` now stores the `Sensor.couplings` graph directly.
- Added `Sensor.get_time_hamiltonian` method which returns the system hamiltonian at a specific time `t`.
- Reworked `Sensor`'s time hamiltonian generation function structure to be more clear.
- `draw_diagram` now scales the linewidth of coupling arrows based on the magnitude of the Rabi frequency.
- Improved accuracy of language regarding rotating frame choices in rydiqule's physics documentation.
- Greatly over-hauled and expanded example notebooks and documentation to cover new features and clarify old ones.
- Added a `Sensor.zip_zips` method to zip axes already containing multiple zipped parameters.
- Updated CyRK timesolver backend to use `pysolve_ivp`. Added an improved differential equation generation method `'flat'` which improves performance by ~30%. This new method is currently not compatible with doppler solves.
- Extended the automated test suite to check docstring examples.
- Added an analytic 1D doppler-averaged steady-state solver `doppler_1d_exact`. This solver is significantly faster for Doppler-averaged solves. For now, this solver is considered experimental.

6.5.2 Bug Fixes

- Fix bug where re-adding a coupling that had a zipped parameter did not invalidate the zip.
- `transition_frequency` is now correctly marked as a non-scannable parameter
- Fixed bugs in `draw_diagram` with un-coupled states and dephasings not toggling correctly.
- Fixed issue where passing the same numpy array to two zipped parameters would result in incorrect tensor broadcasts.

6.5.3 Deprecations

- Overhaul of `Cell` is likely to change results of code that used `Cell` in v1, if not fail outright. Please see documentation for migration guide between v1 and v2.
- Previously deprecated experiment functions have been deleted from `rydiqule.experiments`. These deprecated functions are: `get_transmission_coef`, `get_susceptibility`, `get_phase_shift`, `get_solution_element`, and `get_OD`. Since v1.1.0, this functionality has been incorporated directly into `Sensor_Solution`.
- Internally-used utility functions have been removed from the top-level namespace. All these functions can still be accessed by importing from their sub-module locations. Functions removed from the top-level namespace are `generate_eom`, `get_basis_transform`, `solve_eom_stack`, `generate_eom_time`, `get_doppler_equations`, `generate_doppler_shift_eom`, `doppler_classes`, `doppler_mesh`, `apply_doppler_weights`, `compute_grid`, `matrix_slice`, `memory_size`, `get_slice_num`, and `get_slice_num_t`
- Removed deprecated `Cell.add_states` method.
- `suppress_rwa_warn` kwarg for `Sensor.add_coupling` is deprecated. Now use `warnings.simplefilter('ignore', rq.RWAWarning)` to suppress the warning.
- Renamed `Sensor.get_time_couplings` to `Sensor.get_time_hamiltonian_components`.

- Removed `Sensor.get_time_hamiltonians`. Instead call `Sensor.get_hamiltonian` and `Sensor.get_time_hamiltonian_components` directly.
- `suppress_dipole_warn` kwarg for `Cell.add_coupling` is deprecated. It is no longer possible to add a non-dipole allowed coupling in Cell.
- `Solution` object is no longer a bunch/dict object.
- Dropped support for numba-only timesolver backends.
  - `numbakit-ode` was never much of an improvement, if any for our types of problems
  - `nbrk_ode` (and it's modern replacement `nbsolve_ivp`) are not actively being maintained by CyRK. They also have not provided significant improvements for our types of problems.

6.6 v1.2.3

- Minor hotfix release to pin down incompatible versions of numpy and cyrk dependencies.

6.7 v1.2.2

6.7.1 Improvements

- Now also distribute rydiqule via an anaconda channel.

6.7.2 Bug Fixes

- Fixed bug where t=0 time-dependent hamiltonians calculated in `solve_steady_state` were double counted if more than one time-dependent coupling was present.

6.8 v1.2.1

6.8.1 Bug Fixes

- Fixed bug in energy level shifts where shifts overwrote detunings instead of adding.

6.9 v1.2.0

6.9.1 Improvements

- Level diagrams now use `Sensor.get_rotating_frames` to provide better plotting of energy ordering of levels.
- Level diagrams now allow for optional control of plotting parameters by manually specifying `ld_kw` options on nodes and edges.
- Added the ability to specify energy level shifts (additional Hamiltonian diagonal terms) not accounted for by the coupling infrastructure.

6.9.2 Bug Fixes

- `Sensor.make_real` now returns correct sized `const` array when ground is not removed.
- Many updates to type hints to improve their accuracy.

6.9.3 Deprecations

- Remove `Solution._variable_parameters` in favor of property checking the observable parameters.
- Renamed `Sensor.basis()` and `Solution.basis` to `Sensor.dm_basis()` and `Solution.dm_basis` to disambiguate physical basis from computational basis.

6.10 v1.1.0

6.10.1 Improvements

- Added the ability to specify hyperfine states in a `Cell`. They are distinguished by having 5 quantum numbers `[n, l, j, f, m_f]`.
- `kappa` and `eta` are now properties of `Cell` which are calculated on the fly.
- Separated rotating frame logic from hamiltonian diagonal generation into a new function `Sensor.get_rotating_frames()`. Allows for simple inspection of what rotating frame rydiqule is using in a solve.
- Reworked the under-the-hood parameter zipping framework. This should have minimal impact on user-facing functionality.
  - Hamiltonians with zipped parameters are no longer generated with a `diag` operation.
  - Zipped parameters are now handled with a dictionary rather than a list.
  - Zipped parameters can now be given a shorthand label rather than the default behavior of concatenating individual labels.
- The rearrangement of axes in a stack is now defined completely by the behavior of `axis_labels()`.
- Added a `diff_nearest` boolean argument to `get_snr`. When true, calculates SNR based on nearest neighbor diff. This is in contrast to the default behavior of taking the difference relative to the first element. One case where this is necessary is when getting SNR vs LO Rabi frequency of a heterodyne measurement.
- Added the ability to label states of a sensor with the `label_states` method. States with a label matching a particular pattern can be accessed with the `states_with_label` function.
- Timesolver now allows for returning doppler-averaged solutions without applying the doppler weight factors. This is mostly useful for internal testing.
- `solve_steady_state` now treats time-dependent couplings as having their $t = 0$ value. Most importantly, this affects the default behavior for timesolve initial condition generation and should limit large transient behavior. This also allows the user to specify if time-dependent couplings should be solved with field on or off in steady-state by altering their $t = 0$ value (eg changing between sin and cos).
- Added unit tests for observables, (susceptibility, optical depth, transmission coefficient, and phase shift).
- All Observables (susceptibility, optical depth, etc) now only require a `Solution` object to run.
- `rq.D1_states` and `rq.D2_states` can now specify the atom via string with any isotope specification (including none)
- `get_snr` now warns if any couplings have time-dependence, which are ignored.
- Zipped parameter labels may now include underscores
- `about` function now conceals the user's home directory by default when printing paths
- Moved level diagram plotting to use an external library

6.10.2 Bug Fixes

- Fixed return units of `get_snr` to actually return in 1s BW. Previously was returning in 1us BW.
- Sign errors when specifying detunings both in and out of the rotating frame have been fixed. All detuning signs now follow the convention that positive = blue detuned from atomic resonance, so long as the couplings are added correctly (ie second state of `states` tuple is always the higher energy one).

- Fixed potential issue in `get_snr` where output results could be overwritten to views of intermediate arrays
- Fixed numerical bugs in observables: phase shift, susceptibility, optical depth, transmission coef. Now unit tested against Steck Quantum Optics notes.
- Ensure that non-dipole-allowed transitions are properly warned about in `Cell.add_coupling` with ARC==3.4

6.10.3 Deprecations

- The new `kappa` and `eta` properties of `Cell` directly calculate from Cell properties.
- Time-solver backends (except scipy) are now optional dependencies that are no longer installed by default. To install them, use the `pip install rydiqule[backends]` command.
- The uncollapsed stack shape can no longer be accessed to avoid confusion.
- Removed the ability to pass additional parameters to `np.meshgrid` through the `get_parameter_mesh` function.
- `get_snr` no longer returns in units of 1us.
- Default timesolver initial conditions no longer assume time-dependent couplings have the value of `rabi_frequency`. It is now `rabi_frequency` times the `time_dependence`.
- Multiple sign errors have been corrected in `Sensor` and `Cell` with regards to detunings. Results that are asymmetric about zero detuning are likely to change. Please ensure all couplings are following correct sign conventions for consistent results (ie second state of `states` tuple has higher energy).
- most of the functions in experiments.py have been moved to become methods of `Solution` class.

6.11 v1.0.0

6.11.1 Improvements

- Steady-state behavior for time-dependent fields (and thus initial conditions for time solves) is now computed as a static value rather than zero (previous behavior).
- Added a flag in `scipy_solve` to specify how to define the right-hand function of the differential equation, to use either loops (the newer method) or list comprehension (the older method).
- Implemented `ruff` linting rules as an action for new PRs to help enforce good coding practices.
- Implemented unit-testing action for new PRs to help automate catching regression bugs.

6.11.2 Bug Fixes

- Fixed a broken uinit test that did not affect package functionality.
- Fixed issue where level diagrams don't draw correctly if all non-zero dephasings are equal.

6.11.3 Deprecations

6.12 v1.0.0rc2

6.12.1 Improvements

- Added a `copy` method to solution.
- Expanded the `Solution` object to include more clear axis labels and the basis of the sensor used.
- Begin hosting public documentation on readthedocs.

6.12.2 Bug Fixes

- Changed an `isinstance` check to `hasattr`, fixing an occasional issue with reloading `rydiqule` in jupyter notebooks.
- Fixed issue where submodules were not installed outside of editable mode.
- Fixed a bug where additional arguments like warning suppression could not be passed to Sensor.add_couplings

6.12.3 Deprecations

6.13 v1.0.0rc1

6.13.1 Improvements

- Added a warning in cell if `add_coupling` is called a dipole-forbidden transition.
- The zip_parameters function can now be called on parameters of different types (e.g. detuning with rabi_frequency)
- The time solver now can call ivp solvers outside its own module. This allows for more quickly using different backend solvers for time-dependent problems.
- Implement timesolver backends based on CyRK's cython and numba ode solvers
- Optimize scipy backend of the timesolver for smaller dimensional problems

6.13.2 Bug Fixes

- Fixed issue where solvers would save doppler axes labels and values even when they are summed over to the solution object
- Fixed a bug where energy level diagrams broke when decoherence rates were scanned.
- Fixed issue where compiled timesolvers could not solve doppler averaged problems.
- Fixed issue where certain doppler solves could not be sliced correctly

6.13.3 Deprecations

6.14 v0.5.0

6.14.1 Improvements

- Add isometric-population meshing option to `doppler_mesh`
- Allow `get_rho_ij` to accept a `Solution` object directly, in addition to solution numpy arrays
- Add `get_rho_populations` helper function to efficiently get the trace of density matrix solutions
- Allow `beam_power` or `beam_waist` to be scanned parameters in a `Cell` coupling
- Add more information to `Solution` objects returned by the solvers
- Allow dephasings to be scannable parameters.
- Updated the framework for scanning parameters to generate relevant lists on the fly
  - Note: This changes the order of axes in a stack. Previously, the axes would be ordered based on the order they were added to the system. They are now ordered based on python's `sort()` applied to a tuple of ((low_state, high_state), parameter_name). As a result, they will be ordered first by lower state, then by upper state, then alphabetically by parameter name (e.g. "detuning", "rabi_frequency") In cases where the code was being used for simulations, this may affect cases where axes were defined specifically by number, and these may need to be updated.
- Added a distinction between stack shapes in steady-state vs time-dependent. For example, a steady-state hamiltonian stack may have shape `(10,1,3,3)` while the time dependent portion may have shape `(1,25,3,3)`.

- Renamed the `ham_slice` function to `matrix_slice` and allowed it to iterate over any number of matrices. - Updated internals of solver functions to use this framework.
- `zip_parameters` function no longer enforces parameters be the same type.

6.14.2 Bug Fixes

- Fixed several issues with parameter zipping functionality producing errors when sensor methods were called multiple times.
- Fixed issue where `get_rho_ij` incorrectly calculated the `rho_00` element
- Allow `Cell.add_coupling` to accept a list of e-field values
- Fixed an bug where specifying a list of `rabi_frequency` in a coupling with `time-dependence` would raise an error when solved
- Fixed issue with dephasing broadcasting preventing hamiltonian slices for large solves

6.14.3 Deprecations

- Removed all `sensor_management` functionality as too difficult to maintain generally and securely.
- Removed the internal `_variable_couplings`, `_variable_parameters`, and `_variable_values` attributes from sensor.

6.15 v0.4.0

6.15.1 Improvements

- Changed the handling of decoherent transitions to be stored on graph edges rather than as a separate attribute.
  - Gamma matrix is now calculated on the fly with the `decoherence__matrix()` method.
  - Decoherent transitions are now added with with the `add_decoherence()` function in `Sensor`.
  - `Cell` now calculates transition frequencies and decay rates automatically and places them on the appropriate graph edges.
- Changed the `Sensor.couplings` attribute from a `nx.Graph` to an `nx.DiGraph`. This has multiple advantages:
  - A less vague definition of detuning convention.
  - Precise definition of energy ordering: couplings now always point from lower to higher absolute energy.
  - More flexibility in decoherence. Decoherent transitions now point "from" one state "to" another rather than just "between" 2 states. This fixes a limitation where gamma matrices no longer must be lower triangular.
- `get_snr()` function in `rq.experiments` now takes `kappa` and `eta` as optional arguments to allow for running on any `Sensor` object. They can still be inferred from a `Sensor` subclass that has them as attributes if unspecified.
- time solver now properly handles complex time dependencies in the rotating wave approximation
- Added type hints to code base that can be used to static type check with mypy
- Added functions `rq.calc_kappa` and `rq.calc_eta` to properly calculate kappa and eta constants for experimental parameters.
- Added function `rq.get_OD` that calculates the optical depth of a solution
- Improved accuracy of the solver memory estimates
- Increased input validation unit test coverage

- Generalized handling of transit broadening to allow for multiple repopulation states with varying branching ratios

6.15.2 Bug Fixes

- Fixed an issue with time dependence in the probe laser
- Modified solver to allow for complex time dependence
- Fixed non-hermitian hamiltonians in time solver
- Fixed error with multiple time-dependencies in time solver
- Added functionality to solver error with complex time dependencies
- Modified experimental return functions (`get_transmission_coef()`, `get_phase_shift()`, and `get_susceptibility()`\`) to allow scanning of probe rabi frequency
- Fixed `get_rho_ij` so that it correctly calculates the `(0,0)` population element
- Fix error in `test_sensor_management` which fails if temporary directory does not exist.
- Tighten `test_decoherences` tolerances to the 2pi*100Hz level to catch errors in decoherence matrix generation.
- Fixed issue where `get_snr` ignored the optical path length input parameter
- Fixed issue where calling `solve_steady_state` with `sum_doppler=False` would double memory footprint.
- Fixed issue where `solve_steady_state` could be called with `weight_doppler=False` and `sum_doppler=True`.

6.15.3 Deprecations

- `get_snr` no longer allows manually specifying `Sensor.eta` and `Sensor.kappa`, these values must be passed as args for Sensor input
- Removed unused `gamma_transit` argument from Sensor init
- Re-ordered argument list to `Cell.add_coupling` to match order of `Sensor.add_coupling`
- `Sensor.add_fields` has been fully removed and no longer works as a deprecated alias of `Sensor.add_couplings`

6.16 v0.3.0

6.16.1 Improvements

- Expanded documentation
- Removed restrictions on ARC and numpy versions during installation.
- Vectorized equation of motion generation to support prepending axes to a hamiltonian
- Updated the internal mechanism for sensor handling fields of various type
  - Fields are now internally called couplings
  - Fields are specified as either having rabi_frequency or transition_frequency, corresponding to RWA or non-RWA fields
  - Fields are specified as either having detuning or transition_frequency, corresponding to steady-state or time-dependent fields
  - Fields with specific traits can be accessed with the `couplings_with()` function
- Added a feature to save/load sensors/cells

- Implemented NumbaKitODE which considerably speeds up solve_time. This feature can be enabled by setting parameter compile=True of solve_time.
- Improved logic for building diagonal terms of Hamiltonian using NetworkX graph library that allows for diagonal terms to be built from any set of values.
- Generalized doppler averaging to support prepended axes on hamiltonians.
- Improved time solver logic for improved modularity across doppler solving and multivalue parameters.
- Added a feature to draw level diagram
- Seamlessly generate all Hamiltonians from lists of parameters in sensor.
- Added ability to label couplings.
- Added capability to make any coupling time-dependent
- Sped up time solving considerably by simultaneously solving all equations rather than looping.
- Allow for user to specify fields by beam power, beam waist, and electric field, in the Cell framework.
- Solve functions now return a bunch-type object rather than a tuple.
- Added functionality that breaks equations into slices based on memory requirements
- Quantum numbers and absolute energies are now stored on the nodes of a Cell couplings graph
- Cell now adds decay rates and decoherences to the nodes and edges of the Cell couplings graph
- Cell now calculates the gamma matrix in an arbitrary way, and is no longer limited to two laser, ladder schemes
- Added function to calculate sensor SNR with respect to any varied sensor coupling parameter
- Added function to return sensor parameter mesh

6.16.2 Bug Fixes

- Fixed example notebook.
- Fixed issue where doppler averaging breaks if there are uncoupled levels.
- Fixed doppler averaging so that doppler shifts are applied with signs consistent with the hamiltonian.
- Fixed a bug where doppler averaging did not properly solve separately for each doppler class.
- Fixed issue where spatial dimension of doppler averaging is not introspected correctly in the presence of round-off errors.

6.16.3 Deprecations

- All “field” functionality are being deprecated in favor of “coupling”
- The `rf_couplings`, `target_state`, and `rf_dipole_matrix` arguments of `solve_time()`
- All functions relating to sensor.transition_map are deprecated
- Cell now does not accept gamma_excited or gamma_Rydberg as these are always calculated or Sensor can be used with a given gamma matrix
- Cell now does not accept gamma_doppler as Doppler broadening width is given by multiplying the most probable velocity and the laser k-vector

6.17 v0.2.0

Beta release. Contains very large number of backwards-incompatible changes over alpha release.

6.18 v0.1.0

Alpha release. Minimum viable product release that does basic modeling tasks slowly.

CHAPTER SEVEN

PHYSICS DOCUMENTATION

The following pages contain white-ups that explain the theoretical basis for the many operations performed by rydiqule.

7.1 Equations of Motion Generation

7.1.1 Equations of Motion Generation

This document details a few notes about the theoretical operations taking place under the hood in the Rydiqule Modelling package. In particular, we discuss the methods that Rydiqule uses to numerically solve differential equations for density matrices.

Hamiltonian and Rotating Wave Approximation

For a two level atom interacting with an electric field $\mathbf{E}$, the dipole interaction Hamiltonian is,

$$H = \omega \left|e\right\rangle\left\langle e\right| - \mathbf{d}\cdot\mathbf{E}$$

where the Rabi frequency is defined as $\Omega = \mathbf{d}\cdot\mathbf{E}/\hbar$. The electric field is,

$$\begin{aligned}\mathbf{E} &= \mathbf{E}_0\cos(\omega t+\phi)\\ &= \frac{\mathbf{E}_0}{2}(e^{i\omega t}+e^{-i\omega t})\end{aligned}$$

The dipole operator can be written [1]

$$\mathbf{d} = \left\langle g\right|\mathbf{d}\left|e\right\rangle\left(\left|g\right\rangle\left\langle e\right| + \left|e\right\rangle\left\langle g\right|\right)$$

The operator $\left|g\right\rangle\left\langle e\right|$ evolves at frequency $e^{i\omega t}$ under the bare Hamiltonian, so we expand and take the slowing evolving terms (RWA, see [1]).

$$\begin{aligned}H_{\mathrm{RWA}} &= \omega\left|e\right\rangle\left\langle e\right|\\ &- \left\langle g\right|\mathbf{d}\left|e\right\rangle\cdot\frac{\mathbf{E}_0}{2}(\sigma^+e^{-i\omega t}+\sigma^-e^{i\omega t})\end{aligned}$$

where $\sigma^+ = \left|g\right\rangle\left\langle e\right|$ and $\sigma^- = \left|e\right\rangle\left\langle g\right|$

Equations of Motion

The Master equation that governs system dynamics used by Rydiqule is the Linbladian. It is a semi-classical formulation of the Schroedinger equation for use in open quantum systems.

$$\dot\rho = -\frac{i}{\hbar}[H,\rho] - \mathcal{L}$$

This can be written in summation notation (using Kronnecker deltas),

$$\dot\rho_{ij} = -\frac{i}{\hbar}\left(H_{ik}\rho_{kj} - \rho_{ik}H_{kj}\right) + \sum_{m,n}\frac{\Gamma_{mn}}{2}\left(2\delta_{ij}\rho_{mm} - \delta_{mi}\rho_{ij} - \delta_{mj}\rho_{ij}\right)$$

More generally, we can re-write the system of equations as a super-operator,

$$\dot{\rho}_{ij} = R_{ik}\rho_{kj}$$

By re-shaping these equations, using `numpy.reshape()`, we can convert this into a linear set of differential equations in matrix form (see `generate_eom()`). With the re-shaped density vector p, the equations of motion become

$$\dot{p}_l = M_{li}p_i$$

This is a linear set of equations we can easily solve with `numpy.linalg.solve()`.

Our reshaping procedure defines a new computational basis that is, for basis size b,

$$l = b \times j + i$$

For example,

l	ij
0	00
1	10
2	20
3	01
4	11
5	21

For the programmatic code, we need knowledge of this relationship.

Removing the Ground State

The density vector (matrix) is physically constrained, so that the total population is one. This constraint is not included in the equations of motion. This leads to numerical instabilities. The best way to fix this instability is to algebraically remove one of the equations of motion (ie the ground state). To remove the ground state, we apply the constraint

$$\rho_{00} = 1 - \sum_i \rho_{ii}.$$

Writing this in terms of p gives,

$$p_0 = 1 - \sum_x p_{[(b+1)\times x]}$$

We use this to re-write Eq. \ref{eq:master},

$$\dot{p}_l = M_{li}p_i - M_{l0}p_0 + M_{l0}\left(1 - \sum_x p_{[(b+1)\times x]}\right)$$

This is the equation we must implement to remove the ground state.

In the code, we can apply Eq. \ref{eq:groundRemoved} and then we can simply remove the first column of M_{li}. In the code, we implement this transformation by replacing the set of equations M_{li},

$$M_{li}\rho_i \rightarrow (M_{li} + M'_{li})\rho_i + c_l$$

The constant term c is equivalent to the first column of M_{li}.

$$c_l = M_{l0}$$

The term we need to add, M' is

$$M'_{li} = -M_{l0}\sum_x p_{[i=(b+1)\times x]}$$

This can be implemented as the tensor product of two vectors

$$M'_{li} = -M_{l0} \otimes p^*$$

where M_{i0} is just $M[:, 0]$ and $p^* = p_{[j=(b+1)\times x]}$ is a vector of ones and zeros that is generated with list comprehension.

The end result is an equation where each ground state term of the density matrix ρ_{00} is replaced by the sum of all excited states.

Making the Equations Real

Numerically, converting to a real set of equations is important, because it prohibits the buildup of “imaginary populations” in quantum states. In other words, some equations in the equations of motion are physically required to be real, and some are complex. Machine rounding errors causes leakage into the imaginary parts of the populations equation, which is unphysical. Under certain solving conditions the equations are not stable to this buildup. Converting all the equations to real solves the issue.

The equation we want to solve (for the density vector p) is,

$$\dot{p_c} = M_c \cdot p_c + c_c$$

where the $_c$ notation represents that each term is complex.

The change in basis that we implement is shown below in equation and table format,

$$\begin{aligned} \rho_{ii} &\rightarrow \rho_{ii} \\ \rho_{ij} &\rightarrow Re(\rho_{ij}),\ i > j \\ \rho_{ji} &\rightarrow Im(\rho_{ij}),\ i < j \end{aligned}$$

l	real ij	complex ij
0	ρ_{00}	ρ_{00}
1	ρ_{10}	Re(ρ_{10})
2	ρ_{20}	Re(ρ_{20})
3	ρ_{30}	Re(ρ_{30})
4	ρ_{01}	Im(ρ_{10})
5	ρ_{11}	ρ_{11}

We implement this with a transformation matrix U that is unitary up to a scale factor,

$$\begin{aligned} M_r &= U \cdot M_c \cdot U^{-1} \\ c_r &= U \cdot c_c \end{aligned}$$

This matrix is calculated in the `get_basis_transform()` helper function and is subsequently used to transform between the complex and real bases.

Converting Solutions Back to the Complex Basis

Rydiqule’s solutions are kept in its computational basis (i.e. real, with first state removed). Standard observable calculations using this basis are provided by `Solution`. If you would like to convert the solutions back to the complex basis directly, the utility function `sensor_utils.convert_dm_to_complex()` can be used. The `Solution.complex_rho` convenience attribute provides this conversion for solutions.

References

[1]D. A. Steck, *Quantum and Atom Optics*, 0.13.15 ed. (2022).

7.2 Stacking Conventions

7.2.1 Stacking Conventions

For many problems, Rydiqule is designed to implicitly handle multiple possible values for a single parameter. For example, sweeping over a range of detuning values is handled in Rydiqule simply by specifying the value of interest a list or array rather than a single value. This enables a tremendous amount of flexibility in the problems that Rydiqule can solve naturally, but there are some things worth noting about how Rydiqule specifically handles these problems, which are outlined in this document.

Numpy arrays

Typically, python lists are quite slow to perform operations on since they are dynamically sized and typed. This allows tremendous flexibility in what can be put into a list but some problems with how fast elements are accessed from that list and operating on. Numpy arrays were created to address this limitation and Rydiqule makes extensive use of them to make its calculations fast without losing the ease-of-use benefits of a Python interface. Fundamentally, a numpy `ndarray` is a grid of numbers that has dimensionality `m1` by `m2` by `m3` and so on. Numpy routines are written to operate on these arrays very quickly for large numbers of dimensions.

Stacking

While numpy's own way of handling arrays via matrix broadcasting is well-documented, and most of Rydiqule's own functions use the standard numpy conventions, there are some additional assumptions Rydiqule makes when performing these operations that are worth outlining. Fundamentally, Rydiqule thinks about these `ndarray` objects as groups of matrices, meaning that calculations are performed assuming, for example, that an array of shape `(25, 3, 3)` represents 25 3×3 matrices. This is the array that would be generated if a list of 25 values were provided for a detuning value in a 3-level `Sensor`, and that `Sensor`'s `get_hamiltonian` function were called. Rydiqule seamlessly handles all the work of generating those Hamiltonian matrices for each value, and returns a single array object as an output. Similarly, if 2 values are specified as lists of length 25, a single array of shape `(25, 25, 3, 3)` would be returned, with a different 3×3 Hamiltonian matrix for every combination of parameter values, for a total of 625 Hamiltonian matrices. Rydiqule terms this array a "stack" of Hamiltonians, and the "stack shape" are the axes preceding the actual matrix value axes (in this case `(25, 25)`), and is typically, denoted in Rydiqule as `*l` to make clear that it could be any length of set of values depending on the problem.

Hamiltonian generations is created using this convention, and that carries through to generation of equations of motion, and any other quantities that may have a different matrix for each parameter value. A Hamiltonian stack of shape `(*l, 3, 3)` will generate an equation of motion (eom) stack of shape `(*l, 8, 8)`, with all stack dimensions remaining consistent. Rydiqule's internals are, broadly, agnostic to exactly what the dimensions `*l` represent, and work regardless, as long as the dimensions corresponding to the actual quantities are in the expected position at the end.

Parameter Ordering

Given that any number of parameters may be defined as a list, Rydiqule needs a convention to ensure, in the final result, the values represent what is expected and has not been turned around. It is important to Rydiqule's design philosophy that internal variables not by tracked opaquely, and that quantities are, to the extent possible, generated on the fly in a predictable and reproducible way. This begs the questions, which axis corresponds to which value? Suppose the coupling between states 0 and 1 is swept in detuning over 25 values, as is the coupling between states 1 and 2, the stack shape will be `(25, 25)`, but there are some uncertainties. One might assume that the first axis corresponds the the first laser, and the second axis corresponds to the second laser. However, this is not necessarily obvious, and it might be the other way around without a unifying convention. Rydiqule's solution to this problem turns out to be simple: python's `.sort()` function. Since it always orders things according to the same rules, there is a predictable outcome to which axis is which. The parameters are represent by tuples: `((0,1),"detuning")` and `((1,2),"detuning")`. `.sort()` will sort them first by the lower state of a transition, then by the upper state, then alphabetically by the string parameter name (in this case detuning for both).

With this simple convention, Rydiqule makes these arrays consistent across functions. One can be sure that all values will be exactly what is expected and line up properly for all quantities. Hamiltonians, equations of motion, and solutions will all use the same rules. To avoid figuring this out manually for every system, the `Sensor` module contains the `.axis_labels()` method, which returns a list of which axes are which in string form for results interpretation.

Note that the internal functions which calculate these values don't actually care what the axis are, but they do keep them consistent between calculations.

Doppler

It is worth a quick note how Rydiqule handles doppler broadening, because it leverages the same conventions around stacking as other parameter scans, and it may be encountered and cause confusion if you use Rydiqule enough. If doppler is accounted for in a solve, that typically is not invoked until the relevant `solve` function is called. Given a case of `n_doppler` velocity classes in 1 dimension, a new axis will be prepended to the stack, resulting in a `n_doppler` new doppler-shifted Hamiltonian matrices matrix that was previously in the stack. Typically, this is done under the hood, and these other solutions are averaged over before a result is returned, but examining intermediate values may ultimately result in seeing these axes, even if they are not present in the solution that is returned. Importantly, the solver internals are still agnostic to what these preceding doppler axes represent, giving flexibility and allowing a single function to handle all cases. Again, this is an intermediate step that typically does not affect how results are interpreted, it just helps to understand the internals a little better.

7.3 Observables

7.3.1 Observables

Rydiqule can be used to calculate physical outcomes of experiments. These functions get placed into two categories, **Experiments** and **Observables**. Observables are quantities that can be computed directly from a Solution object, with no additional information. These functions can be found as methods of *`Solution`*. Examples of Observables are the susceptibility, optical depth, transmission coefficient, and phase shift of a probing field. Experiments are quantities that require more information. At the time of writing, the only example is the *`get_snr()`* function.

Observable Derivation

Most of the Observables are derived from the optical/electrical susceptibility. Susceptibility is given by the optical polarizability P^+,

$$P^+ = n \langle g| \hat{d} |e\rangle \rho_{eg} = \epsilon_0 \chi E_{tot}/2$$

This equation may be found in Steck Eq. 6.69 [1]. $\langle g| \hat{d} |e\rangle$ is the dipole matrix element, ρ_{eg} is the density matrix element, n is the atomic density, χ is susceptibility, and E_{tot} is to amplitude of the electric field. The factor of two arises from the rotating wave approximation, such that P^+ represents the atomic polarizability in the rotating frame.

Observable Validation

Rydiqule calculates the susceptibility using equivalent equations, but written in terms of atomic constants κ and η (see [2] for definitions). We validate that rydiqule calculates these observable quantities in a manner consistent with a canonical reference, namely Steck's Quantum and Atom Optics notes [1]. The unit tests in test_solution.py test that rydiqule and Steck align, but assume that the density matrix element ρ_{eg} is correct. Validity of density matrix elements is checked in numerous other tests.

Another more stringent test of Rydiqule is comparing the optical depth using two different methods that are both appropriate for a 2-level atom. All losses from an ideal two-level atom arise from scattering from the excited state. This allows one to write the scattering rate in terms of ρ_{ee}, as is done in Steck's Quantum and Atom Optics notes, Eq. 5.273 [1]. Another way to calculate the scattering rate is using the imaginary term of the susceptibility corresponding to the probing field coupling. This is the method Rydiqule uses, and can be found in Eq 6.73 of [1]. This test is implemented in test_solution.py as the `test_OD_with_steck` unit test.

Running just these observable and experiment unit tests can be done by installing the `pytest` dependencies (see *Unit Tests*), and running the following command from the package parent directory.

```
pytest -m experiments
```

User-Defined Observables

With the release of version 2, rydiqule implements observable calculations in a more general way. This was done to properly support observable calculations for a coupling involving sublevels, where the calculated result relies on more than one matrix element.

Any observable quantity $\mathcal{O}$ of a quantum state defined by a density matrix ρ can be calculated using

$$\mathcal{O} = Tr(\rho A)$$

where A is the associated operator for the desired observable. This operation can be performed on any solution using the *Solution.get_observable()* method, where the user provides A as a (generally complex) matrix the same shape as the system Hamiltonian.

In rydiqule, we are typically concerned with the complex susceptibility of the medium chi, whose associated operator is the dipole operator $\hat{d}$. For a general system, this operator is broken into two parts: a scalar quantity d (the dipole moment) that describes the overall strength of the coupling, and relative coupling strengths $\hat{C}$ that describe the relative strength of couplings between sublevels. These relative strengths are things like Clebsh-Gordon coefficients and are represented as a matrix. In rydiqule's implementation, these coefficients are the `coupling_coefficient` parameters. A more detailed description of how these are defined in `Cell`'s implementation is given in *Fine and Hyperfine Structure with Sublevels*. This coefficient matrix can be obtained for any coupling using the *Solution.coupling_coefficient_matrix()* method.

Finally, the *Solution* object has a method *coupling_coefficient_observable()* which automatically obtains the coupling matrix and calculates the associated observable using the above functions. By default, it will use the probe coupling (i.e. first coupling added to a *Sensor*). This functionality is used under the hood to implement rydiqule's standard observable functions *get_susceptibility()*, *get_OD()*, *get_transmission_coef()*, *get_phase_shift()*.

To obtain a different observable, a user can define their own operator and call *Solution.get_observable()*. An example of this is done in the *Simple NMOR examples* notebook. In those calculations, changes in polarization of the probing field are generated by the dipole operator orthogonal to the input probing field. The notebook shows how to obtain the coefficient matrix for the probe, calculates its orthogonal companion, and uses that operator to calculate the observables for polarization and ellipticity rotations.

References

[1] D. A. Steck, *Quantum and Atom Optics*, 0.13.15 ed. (2022)

[2] D. H. Meyer *et. al.*, *Optimal atomic quantum sensing using electromagnetically-induced-transparency readout*, Phys. Rev. A **104** 043103 (2021)

7.4 Doppler Averaging

7.4.1 Doppler Averaging

This document discusses the methods rydiqule uses to implement doppler-averaging of modelling results. As of version 2.1.0, rydiqule now provides two functions for implementing doppler-averaging: *solve_steady_state()* that averages numerically by sampling and *solve_doppler_analytic()* that averages one spatial dimension analytically and the remaining spatial dimensions numerically. The theoretical background is provided for completeness, but is fairly standard. Rydiqule's implementation of this is less obvious in order to optimize computational efficiency by fully leveraging numpy's vectorized operations. The primary goal of this document is to clearly outline, in a single place, the underlying conventions used by rydiqule when doppler averaging.

Assumptions

Rydiqule currently makes implicit assumptions when modelling atomic systems that influence the treatment of doppler averaging. First, rydiqule's equations of motion are single atom, meaning that atom-atom interactions are ignored. Second, rydiqule only solves these equations under the optically-thin assumption, meaning that input parameters do not change. In particular, absorption of the optical fields through an extended ensemble is not considered.

Both assumptions greatly simplify doppler averaging. Specifically, these assumptions allow us to further assume that atoms in different velocity classes do not interact, meaning that doppler averaging entails a simple weighted average of the atomic response at different velocities by the velocity distribution.

Note that rydiqule assumes a three dimensional distribution of velocities, as is the case for a vapor.

Choosing Velocity Classes to Calculate

The primary influence of doppler velocities on the atomic physics is via Doppler shifts of the applied optical fields: defined as $\Delta = \vec{k} \cdot \vec{v}$, where $\vec{k} = 2\pi/\lambda \cdot \hat{k}$ is the k-vector of the the optical field and $\vec{v}$ is the velocity vector of the atomic velocity class in question. The Doppler shift experienced by an atom is due to the sum of Doppler shifts from each component projection of the atomic velocity relative to the k-vector of the field. In practice, the optical fields are often configured such that there is a basis where some of the k-vector components are zero, reducing the number of dimensions that need to be considered. In the simplest case, all optical fields are colinear, meaning all Doppler shifts are due to velocities projected along a single axis.

Choosing which velocity classes to calculate when performing doppler averaging is a fairly complex meshing problem. Our ultimate goal is to numerically approximate a continuous integral in one to three dimensions using a discrete sum approximation. For thermal vapors, where the spread of doppler velocities is large, resulting in Doppler shifts much greater than other detunings, linewidths, or Rabi frequencies in the problem, most velocity classes only contribute minor incoherent absorption to the final result. This allows for much coarser meshes. The difficulty lies in determining which velocity classes, a-priori, can participate in generally narrow coherent processes. This difficulty scales with the number of optical fields in the problem, as each new field increases the number of possible coherent resonances.

In rydiqule, we have striven to keep the specification of Doppler classes fairly flexible, with the constraint that velocity classes along each averaged dimension are the same (ie a rectangular grid). These classes are calculated in the `doppler_classes()` function, which contains options for complete user specification of all classes, as well as some convenience distributions that can be specified via a few parameters.

Maxwell-Boltzmann Distribution

Given that rydiqule gives single-atom solutions, the total atomic response for a given set of parameters is directly proportional to the total number of atoms represented by those parameters. The Maxwell-Boltzmann distribution is used to determine the fraction of the total population that is in each velocity class.

Following the conventions used by the Wikipedia page for the Maxwell-Boltzmann Distribution, the distribution of velocities for an ensemble of three dimensions is

$$f_{\vec{v}(v_x,v_y,v_z)=\left(\frac{m}{2\pi kT}\right)e^{-\frac{m}{2kT}\left(v_x^2+v_y^2+v_z^2\right)}}$$

Here, v_i represents the atomic velocity component along the cartesian axes, k is Boltzmann's constant, T is the ensemble temperature in Kelvin, and m is the atomic mass in kilograms.

This distribution has a number of general properties. To begin, it is normalized such that integrating over all velocities in 3D space will give unity. There are also a few characteristic speeds associated with this 3D distribution: the most probable speed v_p, the mean speed $\langle v\rangle$, and the rms speed v_{rms}.

$$v_p = \sqrt{\frac{2kT}{m}} \tag{7.1}$$

$$\langle v\rangle = \sqrt{\frac{8kT}{\pi m}} \tag{7.2}$$

$$v_{rms} = \sqrt{\frac{3kT}{m}} \tag{7.3}$$

Note that speed is defined as the magnitude of the velocity vector. Also note that all of these quantities are related to each other by simple numerical prefactors. Finally, observe that the distribution above can be easily separated into each cartesian component.

$$f_{\vec{v}=\prod_i \frac{1}{v_p\sqrt{\pi}}e^{-\frac{-v_i^2}{v_p^2}}}$$

Note that the velocity distribution of each spatial component of the velocity is independently normalized. This allows us to readily produce the appropriate weighting distribution for 1, 2 and 3 dimensional averages as needed without having to perform redundant calculations of distributions where the density matrices to be averaged do not depend on a particular v_i. This function is implemented in `gaussian3d()`.

Numerically Averaging Velocity Classes

Given the above assumptions, the doppler average of the density matrix solutions is given by the integral

$$\bar{\rho_{ij}} = \int \rho_{ij}(\vec{v}) f_{\vec{v} d^3 v}$$

This integral is numerically approximated via a finite sum.

$$\bar{\rho_{ij}} \approx \sum_{klm} \rho_{ij}(v_k, v_l, v_m) f_{\vec{v}(v_k, v_l, v_m) \Delta v_k \Delta v_l \Delta v_m}$$

In rydiqule, the weighting function $f_{\vec{v}}$ is implemented in `gaussian3d()`, the volume element $\Delta v_k \Delta v_l \Delta v_m$ is calculated as the product of the gradients along each axis as calculated by `numpy.gradient()` on the specified velocity classes.

We again note that when all k-vectors along a particular axis are zero, $\rho_{ij}(v_k, v_l, v_m)$ is constant along that axis and that axis of the sum can be separated and assumed to sum to unity due to normalization of the weighting distribution along each dimension.

Rydiqule's Implementation (Numeric Method)

Rydiqule's implementation of Doppler averaging is optimized to minimize duplicate calculations and fully leverage numpy's vectorized and broadcasting operations. The general steps of `solve_steady_state()` are as follows:

1. Choose the doppler velocities to use for the mesh in the average.
2. Generate the Equations of Motion (EOMs) for the base zero velocity class using the machinery described in *Equations of Motion Generation*.
3. Generate the part of the EOMs that are proportional to the atomic velocity components v_i. This is done by generating EOMs for the system with all parameters set to zero except for the optical detunings with associated non-zero k-vector components k_i, multiplied by v_P to give the most probable Doppler shifts.
4. Generate the complete set of EOMs for all velocity classes via a broadcasting sum of the base EOMs with the Doppler EOMs multiplied by the normalized velocity classes along each axis. Each non-zero spatial axis that is to be summed over is pre-pended as an axis to the EOM tensor, as described in *Stacking Conventions*.
5. Solve the entire stack of EOMs.
6. Weight the EOMs according to their velocity classes via the Maxwell-Boltzmann distribution and the discrete velocity volume element, as described above.
7. Sum the solutions along the velocity axes.

Internally, rydiqule defines the necessary components for Doppler averaging via three quantities:

- the normalized velocity classes d, provided by `doppler_classes()`
- the most probable speed v_P (in m/s), provided by the user as a class attribute
- the optical k-vector $\vec{k} = 2\pi/\lambda \cdot \hat{k}$ in (Mrad/m), provided for each coupling that has Doppler shifts to be averaged over

This construction has the benefit of allowing for meshes (ie velocity classes) to be defined in a general way relative to the distribution width v_P, making them easily re-usable for any velocity distribution that obeys the Maxwell-Boltzmann distribution.

Analytically Averaging Velocity Classes

Rydiqule's implementation of analytic doppler-averaging follows the propagator method derived in Exact steady state of perturbed open quantum systems by Omar Nagib and Thad Walker. In one spatial dimension, the time evolution of a system is governed by the master equation in the superoperator form

$$\dot{\rho} = \mathcal{L}\rho$$

At steady state, this equation becomes

$$\mathcal{L}\rho = 0$$

Considering a velocity class, v, $\mathcal{L}$ can be divided into two parts:

$$\mathcal{L}\rho_v = (\mathcal{L}_\prime + v\mathcal{L}_\infty)\rho_v = 0$$

where $\mathcal{L}_\prime$ and $\mathcal{L}_\infty$ do not depend on v. A propagator, G_v, is then constructed such that

$$G_v\rho_0 = \rho_v$$

where ρ_0 is the unique steady state solution when $v = 0$. As shown by Nagib and Walker, this propagator G_v is constructed as

$$G_v = \frac{\nVdash}{\nVdash + v\mathcal{L}_0^- \mathcal{L}_1}$$

where $\mathcal{L}_0^-$ is the Drazin inverse of $\mathcal{L}_0$. Suppose that $\mathcal{L}_0^- \mathcal{L}_1$ can be eigendecomposed as

$$\mathcal{L}_0^- \mathcal{L}_1 = \sum_{\lambda=\lambda_1}^{\lambda_N} \lambda r_\lambda l_\lambda^T$$

where r_λ and l_λ are the right and left eigenvectors of $\mathcal{L}_0^- \mathcal{L}_1$, respectively. Then the propagator is decomposed as

$$G_v = \sum_{\lambda=\lambda_1}^{\lambda_N} \frac{1}{1+v\lambda} r_\lambda l_\lambda^T$$

Thus,

$$\rho_v = G_v\rho_0 = \sum_{\lambda=\lambda_1}^{\lambda_N} \frac{1}{1+v\lambda} r_\lambda l_\lambda^T \rho_0$$

Note that this approach extracts all v-dependence into the algebraic prefactor $1/1+v\lambda$. As a result, we can simply integrate analytically over v to compute the ensemble average:

$$\bar{\rho} = \int \rho_v f_v dv = \sum_{\lambda=0} r_\lambda l_\lambda^T \rho_0 + \sum_{\lambda\neq 0} \frac{\sqrt{\pi/2}}{\sqrt{-\lambda^2}\sigma_v} \exp\frac{-1}{2\lambda^2\sigma_v^2}\left(1 + \text{erf}\left[\frac{\sqrt{-\lambda^2}}{\sqrt{2\lambda^2}\sigma_v}\right]\right) r_\lambda l_\lambda^T \rho_0$$

Note that by rydiqule convention, $\mathcal{L}_v$ contains the prefactor $\sqrt{2}\sigma_v$. Additionally, for numeric stability, rydiqule utilizes scipy.special.erfcx. Thus, the equation implemented in *solve_doppler_analytic()* is

$$\bar{\rho} = \sum_{\lambda=0} r_\lambda l_\lambda^T \rho_0 + \sum_{\lambda\neq 0} \frac{\sqrt{\pi}}{\sqrt{-\lambda^2}} \text{erfcx}\left(\frac{-\sqrt{-\lambda^2}}{\lambda^2}\right) r_\lambda l_\lambda^T \rho_0$$

Rydiqule's Implementation (Analytic Method)

Rydiqule's implementation of Doppler averaging is optimized to minimize duplicate calculations and fully leverage numpy's vectorized and broadcasting operations. In the case of one spatial dimension, *solve_doppler_analytic()* computes the doppler-averaged solution as outlined above. In the case of two or three spatial dimensions, *solve_doppler_analytic()* computes the doppler-averaged solution as follows:

1. Choose the doppler velocities to use for the numeric axes in the average.
2. Generate the Equations of Motion (EOMs) for the base zero velocity class using the machinery described in *Equations of Motion Generation*.
3. Generate the part of the EOMs that are proportional to the atomic velocity components in the numeric axes v_i. This is done by generating EOMs for the system with all parameters set to zero except for the optical detunings with associated non-zero k-vector components k_i, multiplied by v_P to give the most probable Doppler shifts.
4. Generate the complete set of EOMs for all numeric velocity classes via a broadcasting sum of the base EOMs with the Doppler EOMs multiplied by the normalized velocity classes along each axis. Each numeric axis is pre-pended as an axis to the EOM tensor, as described in *Stacking Conventions*.
5. Average over the analytic axis at each point on the numeric velocity mesh using the method above.
6. Weight the analytic averages according to their velocity classes via the Maxwell-Boltzmann distribution and the discrete velocity volume element, as described in the numeric method.
7. Sum the solutions along the numeric axes.

Internally, rydiqule defines the necessary components for Doppler averaging via three quantities:

- the normalized velocity classes d, provided by `doppler_classes()`
- the most probable speed v_P (in m/s), provided by the user as a class attribute
- the optical k-vector $\vec{k} = 2\pi/\lambda \cdot \hat{k}$ in (Mrad/m), provided for each coupling that has Doppler shifts to be averaged over

This construction has the benefit of allowing for meshes (ie velocity classes) to be defined in a general way relative to the distribution width v_P, making them easily re-usable for any velocity distribution that obeys the Maxwell-Boltzmann distribution.

Migrating Doppler averaging from v1 to v2

With the release of v2 of rydiqule, how the user provides the above quantities has changed for both `Sensor` and `Cell`.

In v1, the `'kvec'` parameter of the coupling was defined as the most probable Doppler shift vector (ie $\vec{k} * v_P$). This has been changed in v2 such that `'kvec'` is now defined as the optical k-vector only (in units of Mrad/m), and v_P is provided separately at `Sensor` instantiation or by manually updating the `vP` class attribute. Put simply, moving `Sensor` simulations from v1 to v2 means no longer multiplying the k-vector by v_P, and providing the `vP` attribute.

For `Cell`, v1 code followed the same old convention. Now that `Cell` has improved *ARC integration*, couplings in `Cell` take the `'kunit'` argument which defines the unit propagation axis only. The v_P and $2\pi/\lambda$ factors are calculated automatically and applied to any coupling with `'kunit'` defined.

7.5 Time-Dependence

7.5.1 Time-Dependence

This document discusses how rydiqule implements time-dependent couplings between states. It discusses the how to define these couplings in terms of the relevant coupling parameters as well as some theoretical considerations when working in the rotating wave approximation.

Time-Dependent Couplings

The general form for a time-dependent field is

$$A(t)\cos(\omega(t)t + \phi(t))$$

As it will be helpful later, we will break each time-dependent component into relevant static and time-dependent parts. So $A(t) \equiv A_0(1 + A_t(t))$, $\phi(t) \equiv \phi_0 + \phi_t(t)$, and $\omega(t) \equiv \omega_0 + \Delta + \omega_t(t)$. Note that we have made an explicit choice to allow for a static detuning relative to a static transition frequency ω_0.

Rydiqule Coupling Parameters for Time dependence

When defining a coupling in rydiqule, there are four parameters that define the time-dependence, all specified using specific keys in the coupling dictionary.

1. **The amplitude scale factor**: `'rabi_frequency'` or `'e_field'`. A constant scalar that multiplies the time-dependence function. Only one can be defined. If `'e_field'` is defined, the corresponding dipole moment is used internally to convert it to a Rabi frequency. Rydiqule will automatically apply the factor of 1/2 from the RWA when building the Hamiltonian.
2. **The normalized time dependence function**: `'time_dependence'`. A python function that takes a single argument, t. It is normalized such that a value of 1 corresponds to the field amplitude set by `'rabi_frequency'` or `'e_field'`.
3. **The static detuning from the transition resonance**: `'detuning'`, A constant scalar that defines a fixed detuning relative to the transition frequency. When set, this implicitly defines the coupling in the rotating frame defined by the field frequency $\omega_0 + \Delta$ with the Rotating Wave Approximation applied. Rydiqule's convention is that positive detunings represent a blue detuning relative to the atomic transition (ie photon has more energy than the energy difference between the levels).
4. **or the transition frequency**: `'transition_frequency'`. A constant scalar that defines the atomic transition frequency. When set, this implicitly defines the coupling to *not* be in a rotating frame. As such, the time-dependence will need to use the full field frequency.
5. **The static phase offset**: `'phase'`. A constant scalar that results in a factor of $e^(i\phi)$ applied to the amplitude scale factor for fields defined in the rotating frame. Note that this scale factor can be incorporated directly into the `rabi_frequency`, which is allowed to be a complex number. This factor *cannot* be directly incorporated into alternate Rabi definitions of `Cell` such as `e_field`, which is why this parameter exists. Rydiqule defaults to a phase of 0 for all couplings in the rotating frame (i.e. `detuning` is defined) if phase is not provided. An error is raised when phase is defined outside the rotating frame.

Defining the time-dependence in this way allows us to efficiently construct the time-dependent equations of motion (EOMs) as an expansion of EOMs proportional to each time-dependent function. If we let M_i be an EOM tensor, A_i the amplitude scale factor, Δ_i the detuning, and $f_i(t)$ the time-dependent function, we can express this expansion as

$$M_{tot} = M_0(\Delta_i, ...) + \sum M_i(A_i) \cdot f_i(t)$$

Note that M_0 represents the steady-state EOMs which includes the static detunings for all of the couplings, time-dependent or not.

Example Time-Dependencies

We now provide a few examples of how to write a time-dependent field coupling into the parameters exposed by rydiqule.

RF Heterodyne

In this situation, we want to couple a single transition with two fields with a small detuning between them. One field is the local oscillator (LO), which has constant amplitude, frequency, and phase. It is detuned from the transition resonance by Δ_{LO}. The second field is the signal (S), and is detuned from the LO by a frequency δ_S. It's amplitude is fixed and smaller than the LO amplitude. The phase and frequency of this field is fixed.

One can write this total field as

$$E_{tot} = E_{LO} + E_S = E_{LO}\cos((\omega_0 + \Delta_{LO})t) + E_S\cos((\omega_0 + \Delta_{LO} + \delta_S)t)$$

To convert to rydiqule parameters, we first move to the rotating frame defined by the field frequency $\omega = \omega_0 + \Delta_{LO}$. Mathematically this is done by multiplying the total field by $e^{-i\omega t}$ and dropping the fast-rotating terms (i.e. that have components like $e^{-i2\omega_0 t}$). We then separate the constant amplitude prefactor from the normalized time-dependence.

$$E_{tot-rwa} = \frac{E_{LO}}{2}\left(1 + \frac{E_S}{E_{LO}}e^{i\delta_S t}\right)$$

Note that the rotating wave approximation has discarded the counter-rotating term from cos, leaving the single complex exponential in the time-dependence.

The rydiqule parameters are now defined as

1. Constant amplitude: E_{LO}
2. Time-dependence: $\left(1 + \frac{E_S}{E_{LO}} e^{i\delta_S t}\right)$
3. Detuning: Δ_{LO}

The constant amplitude does not include the factor of 2 from the rotating wave approximation because rydiqule will automatically add it if the detuning coupling parameter is provided. Note that the detuning parameter corresponds to the diagonal term of the resulting RWA hamiltonian. Formally, this term is found by adding the (negative) rotating-frame frequency to the atomic energy on the diagonal. In this simplified system, that gives $\omega_0 - \omega = -\Delta_{LO}$. Rydiqule handles applying the negative sign internally so the user-specified detuning is Δ_{LO}.

Non-Rotating Frame

In some instances, one may wish to solve for a time-dependent coupling outside the rotating frame approximation. This situation is signaled to rydiqule by not defining the `'detuning'` parameter of the relevant coupling.

As an example, we can consider the rf heterodyne field coupling in Eq. \ref{eq:heterodyne_field}. The rydiqule parameters without the rotating wave approximation would be

1. Constant amplitude: E_{LO}
2. Time-dependence: $\cos((\omega_0 + \Delta_{LO})t) + \frac{E_{LO}}{E_S}\cos((\omega_0 + \Delta_{LO} + \delta_S)t)$
3. Detuning: undefined
4. Transition Frequency: ω_0

Note that in this case, a factor of $1/2$ will not be applied by rydiqule to the amplitude and the `'transition_frequency'` is now a required parameter that will be used on the hamiltonian diagonal.

Frequency Sweep in the Rotating-Frame

In this situation, we want to work in a rotating frame which will remove the bulk of the field's frequency, relaxing the time solver's timesteps. We assume a fixed amplitude and a linear frequency sweep through resonance at a rate b, starting at detuning $-\delta_0$.

This field coupling is written as

$$\Omega(t) = \Omega_0 \cos(\phi(t))$$

where

$$\phi(t) = \int_0^t \omega(\tau) d\tau$$

and $\omega(t) = \omega_0 + bt - \delta_0$.

The field coupling with the integration performed is

$$\Omega(t) = \Omega_0 \cos((\omega_0 - \delta_0)t + \frac{bt^2}{2})$$

Moving to the rotating frame and re-writing to match rydiqule's inputs we have

$$\Omega_{rwa}(t) = \frac{\Omega_0}{2} e^{\frac{ibt^2}{2}}$$

The rydiqule parameters are now defined as

1. Constant amplitude: Ω_0
2. Time-dependence: $e^{\frac{ibt^2}{2}}$
3. Detuning: $-\delta_0$

Static Phase Offsets

When doing time-dependent calculations of multi-photon coherent effects to study steady-state spectra (eg studying response to frequency modulations), it can be helpful to set random static phase offsets to the couplings to help the solution converge to steady-state faster. If this isn't done, you often observe are large transient at $t = 0$ due to all fields being approximately coherent even if their frequencies are different.

The field coupling is

$$\Omega(t) = \Omega_0 \cos((\omega_0 + \Delta)t + \phi_0)$$

This phase offset can be moved to the static amplitude scaling factor, reducing the computational complexity of the time-dependence.

$$\Omega_{rwa}(t) = \frac{\Omega_0}{2} e^{i\phi_0}$$

The rydiqule parameters are now defined as

1. Constant amplitude: Ω_0
2. Time-dependence: undefined
3. Detuning: Δ
4. Constant phase offset: ϕ_0

An equivalent definition would be

1. Constant amplitude: $\Omega_0 e^{i\phi_0}$
2. Time-dependence: undefined
3. Detuning: Δ

Note that this coupling does not have time-dependence and would be solved as a steady-state field by not setting the `time_dependence` coupling parameter.

Closed-Loops

If your system involves a closed-loop of couplings (ie there is a circular coupling path), you have to track the overall phase of the circular path when moving to a rotating frame. In particular, a time-dependent phase will accumulate in the loop if any of the couplings in the loop have non-zero detuning from atomic resonance.

Modelling a diamond scheme in a four-level atom would have the following four couplings.

$$\Omega^{(a)}(t) = \Omega_a \cos((\omega_1 + \delta_a)t + \phi_a) \tag{7.4}$$
$$\Omega^{(b)}(t) = \Omega_b \cos((\omega_2 + \delta_b)t + \phi_b) \tag{7.5}$$
$$\Omega^{(c)}(t) = \Omega_c \cos((\omega_3 + \delta_c)t + \phi_c) \tag{7.6}$$
$$\Omega^{(d)}(t) = \Omega_d \cos((\omega_4 + \delta_d)t + \phi_d) \tag{7.7}$$

The atomic transition frequencies obey the relationship $\omega_1 + \omega_2 - \omega_3 - \omega_4 = 0$, with fields 1 and 4 coupling to the ground state, and fields 2 and 3 coupling the highest excited state. Note that the detunings for each field are defined such that a positive value corresponds to a blue detuning from atomic resonance. The field frequencies obey the relationship $\omega_a + \omega_b - \omega_c - \omega_d - \Delta = 0$, where $\Delta = \delta_a + \delta_b - \delta_c - \delta_d$.

Moving to a rotating frame is a non-unique transformation (ie there are many equally valid choices). This means that the time-dependent phase due to non-zero detuning of any field could be accounted for on any of the field couplings in a self-consistent way. However, rydiqule makes an explicit choice for the rotating frame via its shortest path determination of the graph for each state. Accurate modelling requires writing the couplings in the specific rotating frame chosen by rydiqule.

The basic choice made by rydiqule is to use the shortest path from the lowest index node of the connected sub-graph (typically 0). If there are multiple shortest paths (ie multiple paths with the same shortest length), only one is returned.

Typically it is the first equal-length path traversed by the algorithm. Which one that is depends on internals of python (namely dictionary ordering).

Because of these choices, rydiqule will choose multiple branching paths from the ground state in a closed-loop. In order to correctly define the rotating frame, you must ensure that your couplings are defined in rydiqule such that each path is relative to the ground state. An example can demonstrate this subtlety.

In rydiqule code, the above couplings would be defined as (ignoring time dependence)

```
fa = {'states':(0,1), 'detuning':delta_a, 'rabi_frequency':Omega_a, 'phase':phi_a}
fb = {'states':(1,2), 'detuning':delta_b, 'rabi_frequency':Omega_b, 'phase':phi_b}
fc = {'states':(3,2), 'detuning':delta_c, 'rabi_frequency':Omega_c, 'phase':phi_c}
fd = {'states':(0,3), 'detuning':delta_d, 'rabi_frequency':Omega_d, 'phase':phi_d}

s = rq.Sensor(4)
s.add_couplings(fa,fb,fc,fd)
```

Note that we have set the `fc` coupling with reversed ordering to indicate state 2 has higher energy than state 3. We can use rydiqule to tell us which rotating frames will be chosen by calling `get_rotating_frames()`. This will return

```
{<networkx.classes.digraph.DiGraph at 0x1ef491e1910>: {0: [0],
1: [0, 1],
3: [0, 3],
2: [0, 1, 2]}}
```

Note that state 3 has been defined directly from ground, instead of the path `[0, 1, 2, -3]` as is often done when solving this problem on paper. As a result, we have defined the `fd` coupling to be `(0,3)` instead of `(3,0)` to match this convention. This is important since all states need to rotate in a frame that starts from the same state. If we instead defined coupling `fd` with `'states':(3,0)`, the resulting path for state 3 is `[0, -3]` indicating state 3 is lower in energy than state 0 because all paths must start at 0. Put another way, all coupling `'states'` tuples are assumed to be ordered such that the second state has higher energy than the first.

Rotating the above couplings into rydiqule's default frame is accomplished using the unitary rotation operator

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{-it\omega_a} & 0 & 0 \\ 0 & 0 & e^{-it(\omega_a+\omega_b)} & 0 \\ 0 & 0 & 0 & e^{-it\omega_d} \end{pmatrix}$$

The couplings now in the rotating wave approximation are

$$\Omega_{rwa}^{(a)}(t) = \frac{\Omega_a}{2} e^{i\phi_a} \tag{7.8}$$

$$\Omega_{rwa}^{(b)}(t) = \frac{\Omega_b}{2} e^{i\phi_b} \tag{7.9}$$

$$\Omega_{rwa}^{(c)}(t) = \frac{\Omega_c}{2} e^{-i\phi_c} e^{i(\delta_a+\delta_b-\delta_c-\delta_d)t} \tag{7.10}$$

$$\Omega_{rwa}^{(d)}(t) = \frac{\Omega_d}{2} e^{i\phi_d} \tag{7.11}$$

Note that only coupling `fc` has any time-dependence for these CW fields. Obviously, if any of the fields are not CW, that extra time-dependence will need to be accounted for as described in the above examples in addition to the time-dependence described here.

The rydiqule coupling parameters would be written as (letting $i = [a, b, c, d]$)

1. Constant amplitude: $\Omega_i e^{i\phi_i}$ with `fc` differing by a sign $\Omega_d e^{-i\phi_c}$
2. Time-dependence (coupling `fc` only): $e^{i(\delta_a+\delta_b-\delta_c-\delta_d)t}$
3. Detuning: δ_i with δ_c not actually being inserted on the diagonal of the hamiltonian

An equivalent definition using the `phase` parameter is

1. Constant amplitude: Ω_i
2. Time-dependence (coupling `fc` only): $e^{i(\delta_a+\delta_b-\delta_c-\delta_d)t}$
3. Detuning: δ_i with δ_c not actually being inserted on the diagonal fo the hamiltonian
4. Constant phase offset: ϕ_i, with `fc` differing by a sign $-\phi_c$

7.6 Solving with Atomic Structure

7.6.1 Atomic Structure Primer

Rydiqule's `Cell` class provides an interface to the base `Sensor` master equation generation class that incorporates properties of physical atoms provided by ARC. When using `Cell`, you must provide the quantum numbers of each energy level in the system so that properties of each state (e.g. energies, dipole moments, natural lifetimes, dephasing rates, branching ratios) can be used to define parameters of the model (e.g. Rabi frequencies, decoherence matrices, etc).

In order to define these quantum numbers correctly for the problem at hand, some understanding of atomic physics is required. We briefly outline the most salient details below. At present, Rydiqule only directly supports Alkali atoms (i.e. atoms with a single valence electron) via the `AlkaliAtom` class of ARC. The following discussion is therefore limited to Alkali atom atomic structure.

Due to fine structure, every electronic energy level of an atom has a principle quantum number n, an angular momentum quantum number l, a spin quantum number s (taken to be $1/2$ throughout), and a total electronic angular momentum quantum number j.

Each fine structure state $|n, l, j\rangle$ has magnetic sublevels, each with an associated quantum number m_j giving the projection of j along the quantization axis. These levels are normally degenerate, unless an external field is applied. If it's a magnetic field, the resulting splittings are known as the Zeeman effect. If it's an electric field, the resulting splittings are due to the Stark effect.

Finally, some fine structure states have resolveable structure due to the hyperfine interaction with the atomic nucleus. These hyperfine states have an additional total angular momentum quantum number f. The magnetic sublevels of a hyperfine state are denoted by m_f, which has a similar role and dependencies as m_j in a fine structure state.

In a Rydberg atom, the hyperfine structure of the ground states and first few excited states is resolveable (though often obscured by doppler averaging in a thermal ensemble). As a result, these states are typically defined by $|n, l, j, f, m_f\rangle$. This interaction is much less strong for Rydberg states, so the appropriate quantum numbers to use are the fine structure $|n, l, j, m_j\rangle$. As such, fully modeling a real Rydberg atom can require not only a large number of states ($|n, l, j\rangle$), but they have different relevant quantum numbers (J with or without F) and all have many magnetic sublevels (m_j or m_f). The level diagram of Fig. 7.1 shows this complexity. Within a master equation solver like Rydiqule, each sublevel of each state must be treated as an independent element in the basis. Even seemingly simple atomic systems therefore require a computational basis size on the order of a 100, making solutions computationally difficult.

Fine Structure without Magnetic Sublevels

There is a common approximation used in the field of Rydberg sensing to mitigate the need for simulating such large computational bases. In short, so long as the magnetic sublevels are degenerate, each $|n, l, j\rangle$ state can be modeled as a single level, as shown in Fig. 7.2. In many situations, this is a reasonable approximation and leads to highly accurate models, especially in thermal ensembles where Doppler-averaging washes out what small non-degeneracies there may be due to stray fields.

A subtlety to using this approximation is in defining the dipole moment of a transition. The dipole moment depends on the magnetic sublevels of either state, which are ignored by the approximation. When the sublevels are degenerate, any sublevel with population in it will contribute to an average coupling based on the dipole-allowed transitions. For example, a transition between $|5P_{3/2}\rangle$ and $|50D_{5/2}\rangle$ with a linearly polarized field has allowed couplings between $m_j = \pm 1/2 \rightarrow m_j' = \pm 1/2$, and $m_j = \pm 3/2 \rightarrow m_j' = \pm 3/2$. If all m_j sublevels of $|5P_{3/2}\rangle$ are equally populated, the average coupling strength observed with be the average of dipole moment for each sublevel transition. However, it is common for a single $|m_j|$ to be more populated than the others (due to selection rules in coupling atoms from the ground state to higher excited states), in which case choosing just one dipole moment may be more accurate.

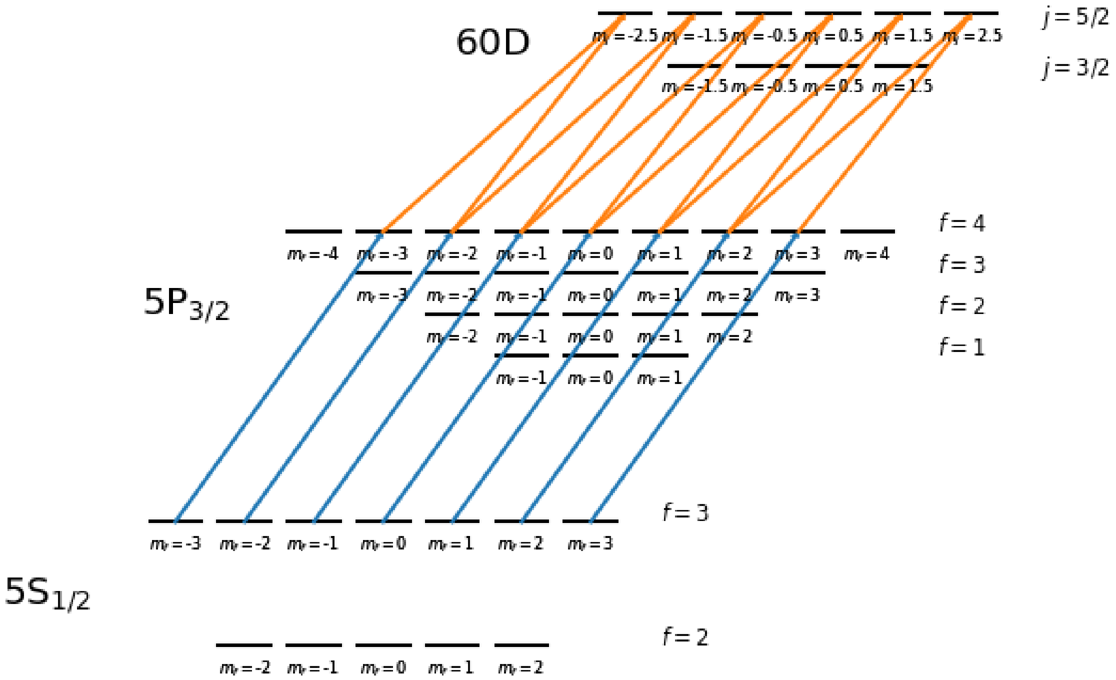


Fig. 7.1: A complete 2-photon Rydberg EIT level structure.

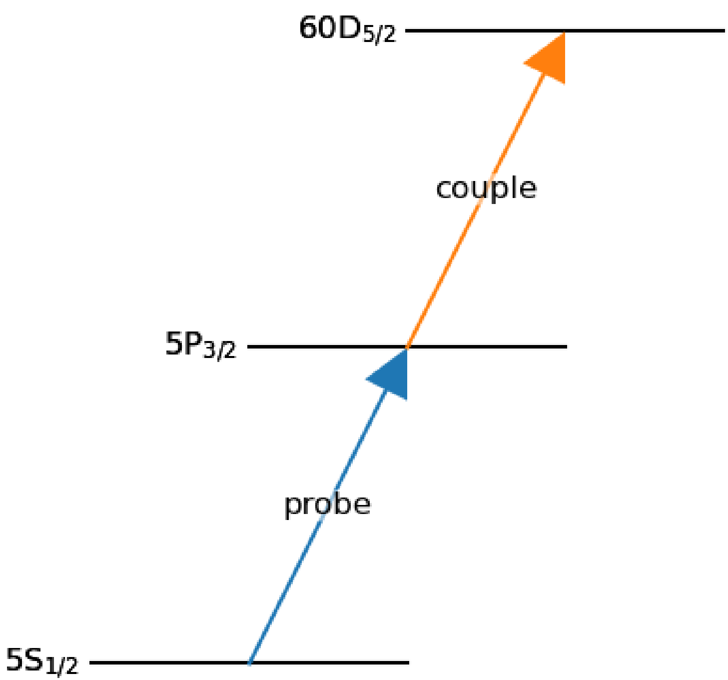


Fig. 7.2: Level diagram of 2-photon Rydberg EIT system using the degenerate sublevels approximation.

Migrating Cell from v1 to v2

This section provides a brief description for how to migrate *Cell* code from v1 to v2 of rydiqule.

Original Rydiqule Behavior

The original implementation of *Cell* solved Rydberg sensing systems under this degenerate approximation. However, it also required specifying m_j for each state. This was only used to define the dipole moment of the couplings, effectively using the single sublevel approximation described above for all transitions. Importantly, specifying this m_j did not strictly model single sublevels (since dephasing rates were only defined by $|n, l, j\rangle$).

Current Rydiqule Behavior

In order to properly support fine and hyperfine structure sublevels in *Cell*, we need to make level specification more accurate to the type of model being solved. To that end, older Rydiqule behavior is specified by only defining states with $|n, l, j\rangle$. The dipole moment of a transition is found by taking the average of the allowed transitions between all the available sublevels (see *get_dipole_matrix_element()* for details). Calculations that relied on the precise value of the dipole moment, such as the *observable functions*, or couplings defined by their electric field (instead of Rabi frequency) are expected to return slightly different results. Otherwise, the behavior will be identical.

Migrating v1 `Cell` code to v2 primarily only involves using *A_QState* to define each state, with only n, l, j defined. There are other smaller differences (such as modified `kvec` definition and other associated ARC automations) that are described in the *Changelog* and *Cell* documentation.

7.6.2 Fine and Hyperfine Structure with Sublevels

In many situations, accurate modeling requires solving for the entire atomic structure (i.e. explicit handling of the magnetic sublevels). These situations include calculating changes in probing polarization and ellipticity angles, non-degenerate states due to ambient magnetic fields, or inhomogeneous coupling strengths due to sublevel structure. As described in *Atomic Structure Primer*, each sublevel must be treated independently within the basis, leading to much larger basis sizes and an associated increase in the complexity of defining the system. With the release of rydiqule v2, the ability to handle calculations involving sublevels has been greatly improved, and this document will discuss those features and the associated physics conventions we use.

Sublevels in Sensor

In *Sensor*, states can be defined using arbitrary tuples, and groups of states can be readily specified by using nested lists within the tuple. Groups of "states" readily represent a manifold of sublevels.

For example, the following code creates a system with four states: a single ground state and a manifold of three excited states. It then adds a coupling from the ground to each excited state.

```
g = (0, 0)
e = (1, [-1, 0, 1])
s = rq.Sensor([g, e])
s.add_coupling((g, e), detuning=1, rabi_frequency=2, label='probe')
```

Here we see that the excited state manifold can be specified by a single object (representing 3 distinct sublevel states), both at `Sensor` creation, and when applying couplings (as well as decoherences).

Another key feature is that the couplings and decoherences can define `coupling_coefficients` which allow for scaling prefactors to be applied to a Rabi frequency for each "sublevel" coupling in the manifold. Modifying the above example

```
g = (0, 0)
e = (1, [-1, 0, 1])
(e1, e2, e3) = rq.expand_statespec(e)

cc = {
      (g,e1): 1/sqrt(2),
```

(continues on next page)

(continued from previous page)

```
        (g,e2): 0,
        (g,e3): -1/sqrt(2)
}

s = rq.Sensor([g, e])
s.add_coupling((g, e), detuning=1, rabi_frequency=2, coupling_coefficients=cc,␣
↪label='probe')
```

The above represents a typical $F = 0 \rightarrow F' = 1$ hyperfine transition that is probed with linearly polarized light, in a quantization axis that is aligned with the optical propagation axis.

Sublevels in Cell

In `Cell`, we combine the features of `Sensor` with a specialized named tuple class to track quantum numbers of states (`A_QState`) and ARC integration wrapped by an internal `RQ_AlkaliAtom` interface to enable automatic calculation of many atomic parameters directly from `A_QState` state specifications.

State Definition

We support three bases for defining the atomic states using `A_QState`:

- NLJ: which averages over sublevel structure, described *here*
- FS: the fine structure basis, where J and m_J are good quantum numbers
- HFS: the hyperfine structure basis, where f and m_f are good quantum numbers

In each case, n, l, j are mandatory arguments in the definition of the `A_QState`. Providing the `'all'` argument to the other parameters will instruct `Cell` to expand the allowed fine or hyperfine states. for example, to define the entire D2 hyperfine transition structure in `Cell`

```
g = rq.A_QState(5, 0, 0.5, f='all', m_f='all')
e = rq.A_QState(5, 1, 1.5, f='all', m_f='all')
c = rq.Cell('Rb85', [g,e])
```

Note

While it is simple to define large atomic bases this way, the hamiltonian size grows very quickly when using sublevels. This is especially true when setting `f='all'`. Be sure your model actually needs all these levels.

Coherent Coupling Definition

We support four classes of transitions between states in these bases:

- NLJ → NLJ
- FS → FS
- HFS → HFS
- HFS → FS (and the inverse)

Note that we can perform models where different bases are used to describe different states, namely HFS and FS. This is particularly useful for Rydberg atoms, where the ground states are best described in the hyperfine basis, but Rydberg states are best described in the fine structure basis. An example of a typical, simplified definition would be

```
g = rq.A_QState(5, 0, 0.5, f=3, m_f='all')
i = rq.A_QState(5, 1, 1.5, f=4, m_f='all')
r = rq.A_QState(50, 2, 2.5, m_J='all')
c = rq.Cell('Rb85', [g,i,r])
```

(continues on next page)

(continued from previous page)

```
c.add_coupling((g,i),
    beam_power=5e-6, # watts
    beam_waist=200e-6, # m, 1/e^2
    detuning=0, q=0, label='probe')
c.add_coupling((i,r),
    beam_power=50e-3,
    beam_waist=180e-6,
    detuning=0, q=0, label='couple')
```

Rydiqule is handling quite a bit automatically here. First, it assumes a gaussian beam profile of waist `beam_waist` and total power `beam_power` to calculate the field strength, which is then used to calculate Rabi frequencies between the outer product of all possible sublevels between the two manifolds. Dipole-allowed transitions will have the associated quantities saved to the graph:

- `dipole_moment`: the transition dipole moment, in units of $a_0 e$
- `coherent_cc`: the angular part of the dipole moment, used to scale the base Rabi frequency, in units of $\langle J||d||J'\rangle/2$
- `rabi_frequency`: the reduced Rabi frequency, i.e. $E \cdot \langle J||d||J'\rangle/2\hbar$

The transition Rabi frequency is given by `rabi_frequency*coherent_cc`. The reason for breaking this up is because the `coherent_cc` (which are at least proportional to Clebsch-Gordon coefficients) are used when calculating observables to properly weight various density matrix components corresponding to a single field. Our convention of defining the base `rabi_frequency` relative to the reduced J matrix element ensures that a common Rabi frequency can be defined for a field spanning many manifolds of sublevels. The functions that calculate these quantities in rydiqule are *get_dipole_matrix_element()*, *get_reduced_rabi_frequency()*, *get_reduced_rabi_frequency2()*, and *get_spherical_dipole_matrix_element()*. Details are given below on how the reduced Rabi frequency, spherical matrix element, and total dipole matrix element are defined.

Note

NLJ transitions use a slight different specification internally. While the angular part is well defined, rydiqule does not use it since there are no other states to meaningfully compare against. For NLJ states, we instead enforce `coherent_cc=1` and the saved Rabi frequency is the full Rabi frequency.

There are alternate methods of specifying the coupling strength in `Cell`. In all cases, the `dipole_moment`, `coherent_cc`, and `rabi_frequency` are defined the same way.

- The first is the `beam_power`/ `beam_waist` definition used above. The function *gaussian_center_field()* calculates the field amplitude at the center of the gaussian spatial mode. Then *get_reduced_rabi_frequency2()* calculates the reduced Rabi frequency.
- The second is by providing the `e_field` directly, which is largely equivalent the above `beam_power`/ `beam_waist` definition, but does not require assuming a gaussian profile (primarily used for RF transitions between Rydberg states).
- The third is by providing the `rabi_frequency` directly. In this case, rydiqule will assume the reduced Rabi frequency has been provided, and calculate the spherical matrix elements accordingly.

Incoherent Coupling Definition

Much like coherent couplings, decoherences between states can be specified between manifolds, using a similar base value/coefficient paradigm. In *Cell*, however, all decoherences due to state natural lifetimes are automatically calculated from the provided basis states. This is done by leveraging ARC to automatically calculate the natural lifetimes of each state as well as the dephasing rates between all states provided in the system definition. These calculations are provided by *get_state_lifetime()* and *get_transition_rate()*.

Of course, it is common to not provide all possible states that every state could decay to. In this situation, rydiqule has three configurable methods for dealing with discrepancies between the natural lifetime of the state and the total sum

of dephasing rates out of a state. Selecting between these options is controlled by the `gamma_mismatch` argument to `Cell`.

1. `'ground'`: Send extra dephasing to the ground state. Here the "ground state" is defined as all atomic states defined in the system that share the same `n,l,j` quantum numbers with the lowest energy in the system. This is rydiqule's default behavior.
2. `'all'`: Proportionally scale existing dephasings so the sum matches the natural lifetime.
3. `'none'`: Ignore the discrepancy.

Dipole Matrix Element Definitions

Here we define how the dipole matrix elements are defined for different kinds of transitions. In each case, we follow ARC's general model of using the Wigner-Eckart theorem to divide the dipole matrix element into angular and radial parts. In particular, we reference all calculations to the symmetric reduced matrix element J for the transition $\langle J||d||J'\rangle$. It is calculated by `getReducedMatrixElementJ()`. It only depends on `n,l,j`, has no angular dependence, and is the common element for all types of transitions `Cell` supports.

Note that we use a slightly different convention for defining the spherical dipole matrix element and reduced matrix element than what ARC uses internally. Namely, we move a factor of two off the reduced matrix element onto the spherical matrix element. This results in the `coherent_cc` parameter being closer to 1, making the Rabi frequency more natural to define. This convention is chosen merely for convenience. The final Rabi frequency in the hamiltonian is identical to what ARC provides.

NLJ

NLJ dipole matrix elements are defined as the average magnitude of all dipole-allowed transitions between sublevels of the two manifolds:

$$d_{\text{NLJ}} = \frac{1}{N}\sum_{m_j=-j}^{j} |\langle n,l,j,m_j|d|n',l',j',m_j+q\rangle|$$

Here, N is defined as the number of non-zero elements in the sum. The spherical matrix element is simply defined as the total matrix element divided by the reduced matrix element in the J basis.

$$s_{\text{NLJ}} = d_{\text{NLJ}}/\langle J||d||J'\rangle$$

This is equivalent to taking the average of the Clebsch-Gordon coefficients for each dipole-allowed transition.

Note

While the spherical matrix element is well defined, rydiqule does not use it in `Cell` since there are no other states to meaningfully compare against. For NLJ states, we instead enforce `coherent_cc=1` and the saved Rabi frequency is the full Rabi frequency.

FS to FS

Fine structure dipole matrix elements are calculated using `getDipoleMatrixElement()` and the spherical matrix element is calculated using and `getSphericalDipoleMatrixElement()` with arguments `j,m_j,j',m_j',q`.

The functional definition (using Wigner3J symbols) is

$$d_{\text{FS}} = s_{\text{FS}}\cdot\langle j||d||j'\rangle \; = (-1)^{j-m_j}\begin{pmatrix} j & 1 & j' \\ -m_j & -q & m_j' \end{pmatrix}\langle j||d||j'\rangle$$

The Clebsch-Gordon coefficients are related to the spherical matrix element by

$$\langle j',m_j';1,q|jm_j\rangle = \sqrt{2j+1}\cdot s_{\text{FS}}$$

Note

Rydiqule's convention is for

HFS to HFS

Hyperfine structure dipole matrix elements are calculated using `getDipoleMatrixElementHFS()`. The spherical matrix element is calculated using `getSphericalDipoleMatrixElement()` (using arguments `f,m_f,f',m_f',q`) and `_reducedMatrixElementFJ()` (which gives the reduced F matrix element in terms of the reduced J matrix element).

The functional definition is

$$d_{\text{HFS}} = s_{\text{HFS}} \cdot \langle nlj||d||n'l'j'\rangle \; = (-1)^{f-m_f} \begin{pmatrix} f & 1 & f' \\ -m_f & -q & m_f' \end{pmatrix} \langle nljf||d||n'l'j'f'\rangle$$

where the reduced F matrix element is defined as

$$\langle nljf||d||n'l'j'f'\rangle \; = (-1)^{j+I+f'+1}\sqrt{(2f+1)(2f'+1)} \; \begin{Bmatrix} f & 1 & f' \\ j' & I & j \end{Bmatrix} \; \langle nlj||d||n'l'j'\rangle$$

Clebsch-Gordon coefficients for these transitions are related to the spherical matrix element by

$$\langle f', m_f'; 1, q|f m_f\rangle = \frac{(-1)^{j+I+f'+1}}{\sqrt{2f'+1}\begin{Bmatrix} f & 1 & f' \\ j' & I & j \end{Bmatrix}} s_{\text{HFS}}$$

Note

Rydiqule's convention is for

HFS to FS

Dipole matrix elements between fine and hyperfine structure sublevels are calculated using `getDipoleMatrixElementHFStoFS()`. The spherical matrix element is calculated using `getSphericalMatrixElementHFStoFS()`.

The spherical part is calculated by expanding the fine basis state into its hyperfine components and summing the elements weighted by Clebsch-Gordon coefficients.

$$s_{\text{HFS-FS}} = \sum_{f'} \langle j', m_j'; I, m_I|f' m_f'\rangle \cdot s_{\text{HFS}}$$

Note

For these transitions, our definition of reduced Rabi frequency can result in `coherent_cc>1`, in similar situations as FS to FS transitions.

CHAPTER

EIGHT

API DOCUMENTATION

rydiqule	Parent computational module.

8.1 rydiqule

Parent computational module.

Modules

arc_utils	Helper methods for interfacing with ARC atom classes
atom_utils	Utilities for interacting with atomic parameters and ARC.
cell	Subclass of `Sensor` with functionality for representing real atoms.
doppler_exact	Steady-state solver for analytical Doppler averaging
doppler_utils	Utilities for implementing Doppler averaging
exceptions	Rydiqule Custom Exceptions and Warnings
experiments	Standard methods for converting results to physical values.
rydiqule_utils	General rydiqule package utilities
sensor	Sensor objects that control solvers.
sensor_solution	Object used to store aspects of a solution when calling rydiqule.solve() Adds essential keys with "None" entries
sensor_utils	Utilities used by the Sensor classes.
slicing	
solvers	Steady-state solvers of the Optical Bloch Equations.
stack_solvers	
timesolvers	Solvers for time domain analysis with an arbitrary RF field

8.1.1 rydiqule.arc_utils

Helper methods for interfacing with ARC atom classes

Classes

RQ_AlkaliAtom(arc_atom)

rydiqule.arc_utils.RQ_AlkaliAtom

class rydiqule.arc_utils.**RQ_AlkaliAtom**(*arc_atom: arc.alkali_atom_functions.AlkaliAtom*)

Bases: `object`

__init__(*arc_atom: arc.alkali_atom_functions.AlkaliAtom*)

Rydiqule's wrapper class around ARC's Alkali atom classes.

Designed predominantly for internal use, seldom needs to be accessed directly.

Parameters

arc_atom (`AlkaliAtom`) – ARC atom to use for calculations. Stored internally in the `atom` attribute.

Methods

__init__(arc_atom)	Rydiqule's wrapper class around ARC's Alkali atom classes.
gaussian_center_field(laserPower, laserWaist)	Returns the electric field for the center of a TEM00 gaussian spatial mode
get_dipole_matrix_element(state1, state2, q)	Get dipole matrix element $\langle s1 \vert e\mathbf{r} \vert s2 \rangle$ in units of $a_0 e$
get_rabi_frequency(state1, state2, q, ...[, s])	Returns the Rabi frequency for resonantly driven atom in center of a TEM00 mode of a field.
get_rabi_frequency2(state1, state2, q, ...)	Returns the Rabi frequency for resonantly driven atom in a given electric field amplitude.
get_reduced_matrix_elementJ(state1, state2)	Returns the reduced dipole matrix element in the J basis.
get_reduced_rabi_frequency(state1, state2, ...)	Returns the Rabi frequency for resonantly driven atom in center of a TEM00 mode of a field.
get_reduced_rabi_frequency2(state1, state2, ...)	Returns the reduced Rabi frequency for resonantly driven atom in a given electric field amplitude.
get_spherical_dipole_matrix_element(state1, ...)	Returns the spherical part of the dipole matrix element for a transition.
get_state_energy(state[, s])	Returns the energy of the level relative to the ionisation level in Hz.
get_state_lifetime(state[, temperature, ...])	Get lifetime of the state.
get_transition_frequency(state1, state2[, ...])	Returns the transition frequency (energy difference) between two states, in Hz.
get_transition_rate(state1, state2[, ...])	Returns transition rate between two states due to spontaneous emission.
get_transition_wavelength(state1, state2[, ...])	Returns the transition wavelength between two states, in m.

Attributes

arc_atom	ARC atom with which to perform calculations.

_getDipoleMatrixElementFStoHFS(*n1, l1, j1, mj1, n2, l2, j2, f2, mf2, q, s*)

Funtion which inverts arcs HFS to FS dipole matrix element function, , in units of e*a0

_get_nlj_dipole(*n1, l1, j1, n2, l2, j2, q, s*)

Dipole matrix element between a pair of NLJ states, in units of e*a0

_tr_prefactor_fs_fs(*state1, state2, s*)

transition rate prefactor between 2 hyperfine states. Units of reduced matrix element $< j|er|j' >$

_tr_prefactor_fs_hfs(*state1*, *state2*, *s*)

transition rate prefactor for transitions from a fine structure state to a hyperfine state. Units of reduced matrix element $< j|er|j' >$

_tr_prefactor_hfs_fs(*state1*, *state2*, *s*)

transition rate prefactor for transitions from a hyperfine state to a fine structure state. Units of reduced matrix element $< j|er|j' >$

_tr_prefactor_hfs_hfs(*state1*, *state2*, *s*)

transition rate prefactor between hyperfine states. returns the branching ratio for a particular F state. Units of reduced matrix element $< j|er|j' >$

arc_atom

ARC atom with which to perform calculations.

gaussian_center_field(*laserPower: float*, *laserWaist: float*) → float

Returns the electric field for the center of a TEM00 gaussian spatial mode

This calculates the peak intensity of the gaussian mode, and uses a plane wave assumption to get the electric field amplitude.

Parameters

- **laserPower** (*float*) – laser power in Watts
- **laserWaist** (*float*) – laser $1/e^2$ waist (radius) in meters

Returns

Peak electric field for a gaussian spatial mode, in V/m

Return type

float

get_dipole_matrix_element(*state1: A_QState*, *state2: A_QState*, *q: Literal[-1, 0, 1]*, *s: float = 0.5*) → float

Get dipole matrix element $\langle s1|e\mathbf{r}|s2\rangle$ in units of $a_0 e$

If states 1 and 2 are sublevels, either FS or HFS, appropriate ARC function is used. If states 1 and 2 are NLJ, a simple average of the magnitudes of the dipole-allowed moments between mJ1 and mJ2 is returned.

Cannot calculate dipole matrix elements between states with NLJ specification and those with either FS or HS splitting.

ARC functions used are:

- `getDipoleMatrixElement()`
- `getDipoleMatrixElementHFS()`
- `getDipoleMatrixElementHFStoFS()`

Parameters

- **state1** (*A_QState*) – `A_QState` namedtuple of quantum numbers for the quantum state $s1$.
- **state2** (*A_QState*) – `A_QState` namedtuple of quantum numbers for the quantum state $s2$.
- **q** (*int*) – Polarization of coupling field in spherical basis (+1, 0, -1), corresponding to σ^+, π, or σ^-.
- **s** (*float, optional*) – total spin angular momentum of the state. Default is 0.5 for Alkali atoms.

Returns
Dipole moment of the transition in atomic units ($a_0 e$). Will be 0 if the transition is not dipole-allowed.

Return type
float

Raises
AtomError – If the two states to be coupled are in one each of the NLJ and FS/HFS definitions.

Examples

```
>>> import arc
>>> g_nlj = rq.A_QState(5, 0, 0.5)
>>> g_fs = rq.A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = rq.A_QState(5, 1, 0.5)
>>> e_hfs = rq.A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_dipole_matrix_element(g_nlj, e_nlj, q=0))
1.7277475900721146
>>> print(my_atom.get_dipole_matrix_element(g_fs, e_hfs, q=0))
1.2217020371187075
>>> print(my_atom.get_dipole_matrix_element(g_nlj, e_hfs, q=0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rydiqule.exceptions.AtomError: Invalid transition type for dipole␣
↪calculation.
```

get_rabi_frequency(*state1: A_QState*, *state2: A_QState*, *q: Literal[-1, 0, 1]*, *laserPower: float*, *laserWaist: float*, *s: float = 0.5*) → float

Returns the Rabi frequency for resonantly driven atom in center of a TEM00 mode of a field.

The field is calculated using `gaussian_center_field()`. It then calls `get_rabi_frequency2()` to get the rabi frequency.

Parameters

- **state1** (*A_QState*) – NamedTuple of quantum numbers for state driving from
- **state2** (*A_QState*) – NamedTuple of quantum numbers for state driving to
- **q** (*int*) – laser polarization in spherical basis (-1,0,1) corresponding to σ^-, π, and σ^+
- **laserPower** (*float*) – laser power in Watts
- **laserWaist** (*float*) – laser $1/e^2$ waist (radius) in meters
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns
rabi_frequency – Rabi frequency in rad/s. To get Hz, divide by 2π

Return type
float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
```

(continues on next page)

(continued from previous page)

```
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_rabi_frequency(g_nlj, e_nlj, q=0, laserPower=1,␣
↪laserWaist=0.01)/1e6) #MHz
304.22
>>> print(my_atom.get_rabi_frequency(g_fs, e_hfs, q=0, laserPower=1,␣
↪laserWaist=0.01)/1e6) #MHz
215.11
>>> print(my_atom.get_rabi_frequency(g_nlj, e_hfs, q=0, laserPower=1,␣
↪laserWaist=0.01))
Traceback (most recent call last):
  ...
rydiqule.exceptions.AtomError: Invalid transition type for dipole␣
↪calculation.
```

get_rabi_frequency2 (*state1: A_QState*, *state2: A_QState*, *q: Literal[-1, 0, 1]*, *electricFieldAmplitude: float*, *s: float = 0.5*) → float

Returns the Rabi frequency for resonantly driven atom in a given electric field amplitude.

Uses `get_dipole_matrix_element()` for the calculation.

Parameters

- **state1** (`A_QState`) – NamedTuple of quantum numbers for state driving from
- **state2** (`A_QState`) – NamedTuple of quantum numbers for state driving to
- **q** (*int*) – laser polarization in spherical basis (-1,0,1) corresponding to σ^-, π, and σ^+
- **electricFieldAmplitude** (*float*) – amplitude of driving electric field, in V/m
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns

rabi_frequency – Rabi frequency in rad/s. To get Hz, divide by 2π

Return type

float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = rq.A_QState(5, 0, 0.5)
>>> g_fs = rq.A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = rq.A_QState(5, 1, 0.5)
>>> e_hfs = rq.A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> e_field = 0.1 #V/m
>>> print(my_atom.get_rabi_frequency2(g_nlj, e_nlj, q=0,␣
↪electricFieldAmplitude=e_field))
13890.429
>>> print(my_atom.get_rabi_frequency2(g_fs, e_hfs, q=0,␣
↪electricFieldAmplitude=e_field))
9822.0166
```

(continues on next page)

(continued from previous page)

```
>>> print(my_atom.get_rabi_frequency2(g_nlj, e_hfs, q=0,␣
↪electricFieldAmplitude=e_field))
Traceback (most recent call last):
  ...
rydiqule.exceptions.AtomError: Invalid transition type for dipole␣
↪calculation.
```

get_reduced_matrix_elementJ(*state1: A_QState*, *state2: A_QState*, *s: float = 0.5*) → float

Returns the reduced dipole matrix element in the J basis.

A convenience wrapper for `getReducedMatrixElementJ()`

Note

To get proper sign conventions, state order must go from lower energy to higher energy state.

Parameters

- **state1** (`A_QState`) – NamedTuple of quantum numbers for lower state
- **state2** (`A_QState`) – NamedTuple of quantum numbers for higher state
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns

Reduced matrix element $\langle J||d||J'\rangle$

Return type

float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_reduced_matrix_elementJ(g_nlj, e_nlj))
4.2321
>>> print(my_atom.get_reduced_matrix_elementJ(g_fs, e_hfs))
4.2321
>>> print(my_atom.get_reduced_matrix_elementJ(g_nlj, e_hfs))
4.2321
```

get_reduced_rabi_frequency(*state1: A_QState*, *state2: A_QState*, *laserPower: float*, *laserWaist: float*, *s: float = 0.5*) → float

Returns the Rabi frequency for resonantly driven atom in center of a TEM00 mode of a field.

The field is calculated using *gaussian_center_field()*. It then calls *get_rabi_frequency2()* to get the rabi frequency.

Note

This function preserves the sign of the dipole moment (i.e. the result could be negative). As such, state calling order matters. To get correct convention for use in Cell, `state` must be lower energy than `state2`.

Parameters

- **state1** (*A_QState*) – NamedTuple of quantum numbers for state driving from
- **state2** (*A_QState*) – NamedTuple of quantum numbers for state driving to
- **laserPower** (*float*) – laser power in Watts
- **laserWaist** (*float*) – laser $1/e^2$ waist (radius) in meters
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns
rabi_frequency – Rabi frequency in rad/s. To get Hz, divide by 2π

Return type
float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_reduced_rabi_frequency(g_nlj, e_nlj, laserPower=1,␣
↪laserWaist=0.01)/1e6) #MHz
372.59
>>> print(my_atom.get_reduced_rabi_frequency(g_fs, e_hfs, laserPower=1,␣
↪laserWaist=0.01)/1e6) #MHz
372.59
>>> print(my_atom.get_reduced_rabi_frequency(g_nlj, e_hfs, laserPower=1,␣
↪laserWaist=0.01)/1e6) #MHz
372.59
```

get_reduced_rabi_frequency2(*state1: A_QState, state2: A_QState, electricFieldAmplitude: float, s: float = 0.5*) → float

Returns the reduced Rabi frequency for resonantly driven atom in a given electric field amplitude.

Uses $1/2$*get_reduced_matrix_elementJ()*.

> **Note**
>
> This function preserves the sign of the dipole moment (i.e. the result could be negative). As such, state calling order matters. To get correct convention for use in Cell, `state1` must be lower energy than `state2`.

Parameters

- **state1** (*A_QState*) – NamedTuple of quantum numbers for state driving from
- **state2** (*A_QState*) – NamedTuple of quantum numbers for state driving to

- **electricFieldAmplitude** (*float*) – amplitude of driving electric field, in V/m
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns
rabi_frequency – Rabi frequency in rad/s. To get Hz, divide by 2π

Return type
float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> e_field = 0.1 #V/m
>>> print(my_atom.get_reduced_rabi_frequency2(g_nlj, e_nlj,␣
↪electricFieldAmplitude=e_field))
17012.23
>>> print(my_atom.get_reduced_rabi_frequency2(g_fs, e_hfs,␣
↪electricFieldAmplitude=e_field))
17012.23
>>> print(my_atom.get_reduced_rabi_frequency2(g_nlj, e_hfs,␣
↪electricFieldAmplitude=e_field))
17012.23
```

get_spherical_dipole_matrix_element(*state1: A_QState*, *state2: A_QState*, *q: Literal[-1, 0, 1]*, *s: float = 0.5*) → float

Returns the spherical part of the dipole matrix element for a transition.

Calculated by dividing the transition dipole moment by the reduced J matrix element.

Parameters

- **state1** (*A_QState*) – NamedTuple of quantum numbers for first state
- **state2** (*A_QState*) – NamedTuple of quantum numbers for second state
- **q** (*int*) – field polarization in the spherical basis (-1,0,1) corresponding to σ^-, π, and σ^+
- **s** (*float, optional*) – total spin angular momentum of the states. By default 0.5 for Alkali atoms.

Returns
Spherical part of the dipole matrix element, in units of reduced matrix element $\langle J||d||J'\rangle$

Return type
float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
```

(continues on next page)

(continued from previous page)

```
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_spherical_dipole_matrix_element(g_nlj, e_nlj, q=0))
0.4082
>>> print(my_atom.get_spherical_dipole_matrix_element(g_fs, e_hfs, q=0))
0.2887
>>> print(my_atom.get_spherical_dipole_matrix_element(g_nlj, e_hfs, q=0))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rydiqule.exceptions.AtomError: Invalid transition type for dipole␣
→calculation. Allowed types are [('HFS', 'FS'), ('FS', 'HFS'), ('HFS',
→'HFS'), ('FS', 'FS'), ('NLJ', 'NLJ')]
```

get_state_energy(*state: A_QState*, *s: float = 0.5*) → float

Returns the energy of the level relative to the ionisation level in Hz.

If `state` is in the fine basis or NLJ, energies are relative to the center of gravity of the hyperfine split states. If `state` is in the hyperfine basis, hyperfine shifts are applied.

Uses ARC's `getEnergy()` to get the energy of the fine structure state. Hyperfine shifts are applied using ARC's `getHFSCoefficients()` and `getHFSEnergyShift()` methods.

Parameters

- **state** (`A_QState`) – `A_QState` namedtuple of quantum numbers of the quantum state for which energy will be calculated.
- **s** (*float*) – Total spin of the state. Default is 0.5 for Alkali atom.

Returns

Energy of state relative to the ionisation level in Hz.

Return type

float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_state_energy(g_nlj)/1e9) #GHz
-1010024.7
>>> print(my_atom.get_state_energy(g_fs)/1e9) #GHz
-1010024.7
>>> print(my_atom.get_state_energy(e_hfs)/1e9) #GHz
-632917.5
```

get_state_lifetime(*state: A_QState*, *temperature: float = 0.0*, *includeLevelsUpTo: int = 0*, *s: float = 0.5*) → float

Get lifetime of the state.

If temperature is provided, includes Black-Body Radiation induced transitions. Otherwise, this is the natural lifetime of the state.

This is a thin wrapper around ARC's `getStateLifetime()` method. It adds basic validation of the state and selects the correct quantum numbers for the calculation.

Parameters

- **state** (`A_QState`) – NamedTuple of quantum numbers of state for which to calculate lifetime.
- **temperature** (*float*, *optional*) – Temperature at which the atom environment is, in Kelvin. Used for calculating the black-body-induced state lifetime. If 0.0 (default), result does not include BBR term.
- **includeLevelsUpTo** (*int*, *optional*) – If `temperature` is non-zero, this specifies the highest principal quantum number states to include in the BBR calculation. Must be at least `n+1` of the provided state.
- **s** (*float*, *optional*) – total spin angular momentum of the state. Default is 0.5, for alkali atoms.

Returns

State lifetime in seconds.

Return type

float

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> e = A_QState(10, 0, 0.5)
>>> e_fs = A_QState(10, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(10, 1, 0.5)
>>> e_hfs = A_QState(10, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_state_lifetime(e_nlj))
1.162e-06
>>> print(my_atom.get_state_lifetime(e_fs))
4.209e-07
```

get_transition_frequency(*state1: A_QState*, *state2: A_QState*, *s1: float = 0.5*, *s2: float = 0.5*) → float

Returns the transition frequency (energy difference) between two states, in Hz.

Uses *get_state_energy()* on both states to determine the energy difference.

Parameters

- **state1** (`A_QState`) – `A_QState` namedtuple of quantum numbers for the first quantum state.
- **state2** (`A_QState`) – `A_QState` namedtuple of quantum numbers for the second quantum state.
- **s1** (*float*, *optional*) – spin of the initial state. Default is 0.5 for Alkali atom.
- **s2** (*float*, *optional*) – spin of the final state. Default is 0.5 for Alkali atom.

Returns

Transition frequency between the two states, in Hz. If negative, state1 has higher energy than state2.

Return type

float

Examples

```
>>> import arc
>>> g_nlj = rq.A_QState(5, 0, 0.5)
>>> g_fs = rq.A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = rq.A_QState(5, 1, 0.5)
>>> e_hfs = rq.A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_transition_frequency(g_nlj, e_nlj))
377107433259213.6
>>> print(my_atom.get_transition_frequency(g_fs, e_hfs))
377107222336963.6
>>> print(my_atom.get_transition_frequency(g_nlj, e_hfs))
377107222336963.6
```

get_transition_rate(*state1: A_QState*, *state2: A_QState*, *temperature: float = 0.0*, *s: float = 0.5*) → float

Returns transition rate between two states due to spontaneous emission.

If temperature is provided, Black-Body Radiation induced transitions are included. Otherwise, rate is due to the natural radiative lifetime only.

Uses ARC's `getTransitionRate()` to get the base transition rate between $|n_1, l_1, j_1\rangle$ to $|n_2, l_2, j_2\rangle$. If state1 and/or state2 are sublevels in the fine or hyperfine structures, this further applies the appropriate branching ratio.

Parameters

- **state1** (`A_QState`) – NamedTuple of quantum numbers of the originating state
- **state2** (`A_QState`) – NamedTuple of quantum numbers of the target state
- **temperature** (*float, optional*) – Temperature of the atomic environment for calculating BBR-induced decays, in Kelvin. With default of 0.0, only include natural lifetime.
- **s** (*float, optional*) – total spin angular momentum. Default of 0.5 for Alkali atoms

Returns
Transition rate in 1/s

Return type
float

Raises
AtomError – If states are in both NLJ and FS/HFS definition.

Examples

```
>>> from rydiqule import A_QState
>>> import arc
>>> g_nlj = A_QState(5, 0, 0.5)
>>> g_fs = A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = A_QState(5, 1, 0.5)
>>> e_hfs = A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_transition_rate(e_nlj, g_nlj)/1e6)  #GHz
36.11
>>> print(my_atom.get_transition_rate(e_hfs, g_fs)/1e6)  #GHz
18.06
```

(continues on next page)

(continued from previous page)

```
>>> print(my_atom.get_transition_rate(e_hfs, g_nlj))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
rydiqule.exceptions.AtomError: Transition between HFS and NLJ are not␣
↪allowed.
```

get_transition_wavelength(*state1: A_QState*, *state2: A_QState*, *s1: float = 0.5*, *s2: float = 0.5*) → float

Returns the transition wavelength between two states, in m.

Uses `getTransitionWavelength()`.

Parameters

- **state1** (`A_QState`) – `A_QState` namedtuple of quantum numbers for the first quantum state.
- **state2** (`A_QState`) – `A_QState` namedtuple of quantum numbers for the first quantum state.
- **s1** (*float, optional*) – spin of the initial state. Default is 0.5 for Alkali atom.
- **s2** (*float, optional*) – spin of the final state. Default is 0.5 for Alkali atom.

Returns

Transition wavelength between the two states, in m. If negative, state2 has higher energy than state1.

Return type

float

Examples

```
>>> import arc
>>> g_nlj = rq.A_QState(5, 0, 0.5)
>>> g_fs = rq.A_QState(5, 0, 0.5, m_j=-0.5)
>>> e_nlj = rq.A_QState(5, 1, 0.5)
>>> e_hfs = rq.A_QState(5, 1, 0.5,f=2, m_f=0)
>>> arc_atom = arc.alkali_atom_data.Rubidium85()
>>> my_atom = rq.RQ_AlkaliAtom(arc_atom)
>>> print(my_atom.get_transition_wavelength(g_nlj, e_nlj))
7.949789146530309e-07
>>> print(my_atom.get_transition_wavelength(g_fs, e_hfs))
7.949789146530309e-07
>>> print(my_atom.get_transition_wavelength(g_nlj, e_hfs))
7.949789146530309e-07
```

8.1.2 rydiqule.atom_utils

Utilities for interacting with atomic parameters and ARC.

Module Attributes

ATOMS	Alkali atoms defined by ARC that can be used with `Cell`.

rydiqule.atom_utils.ATOMS

rydiqule.atom_utils.**ATOMS = {'Cs': 'Caesium', 'H': 'Hydrogen', 'K39': 'Potassium39', 'K40': 'Potassium40', 'K41': 'Potassium41', 'Li6': 'Lithium6', 'Li7': 'Lithium7', 'Na': 'Sodium', 'Rb85': 'Rubidium85', 'Rb87': 'Rubidium87'}**

Alkali atoms defined by ARC that can be used with `Cell`.

Functions

`D1_excited`(n[, splitting, expand])	Retrieve `A_QState` of the excited state of the D1 line of an atom or principle quantum number.
`D1_states`(n[, splitting, g_splitting, ...])	Return the ground and excited states for the D1 line of a rydberg atom.
`D2_excited`(n[, splitting, expand])	Retrieve `A_QState` of the excited state of the D2 line of an atom or principle quantum number.
`D2_states`(n[, splitting, g_splitting, ...])	Return the ground and excited states for the D1 line of a rydberg atom.
`calc_eta`(omega, dipole_moment, beam_area)	Calculates the eta constant needed from some experiment calculations
`calc_kappa`(omega, dipole_moment, density)	Calculates the kappa constant needed for observable calculations.
`expand_qnums`(qstates[, I])	Expand all list-like A_QStates in a list.
`expand_single_qnum`(qstate[, I, wildcard])	Generates a list of all valid states given a particular quantum number to be expanded.
`get_valid_f`(state[, I])	Return the valid values of f for given other quantum numbers.
`get_valid_j`(state[, I])	Return the valid values of j for given other quantum numbers.
`get_valid_mf`(state[, I])	Return the valid values of m_f for given other quantum numbers.
`get_valid_mj`(state[, I])	Return the valid values of m_J for given other quantum numbers.
`ground_state`(n[, splitting, expand])	Retrieve `A_QState` for the ground state of an atom or principle quantum number.
`match_A_QState`(qstate[, compare_list, I])	Function to return all states in a list matching the provided pattern.
`validate_qnums`(qstate[, I])	Validate that the provided named_tuple is a valid rydberg atomic state

rydiqule.atom_utils.D1_excited

rydiqule.atom_utils.**D1_excited**(*n: int | str, splitting: Literal[None, 'fs', 'hfs'] = None, expand: bool = False*) → *A_QState* | List[*A_QState*]

Retrieve `A_QState` of the excited state of the D1 line of an atom or principle quantum number.

Optionally, include fine structure splitting or hyperfine splitting, which will include all `m_j` or all `f` and `m_f` values respectively. By default, specifying splitting does not return a list of states, but rather the associated specification with `"all"` in the appropriate place. The `expand` keyword argument can be used to modify this behavior, returning a list. place.

Parameters

- **n** (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must begin with ['H', 'Li', 'Na', 'K', 'Rb', 'Cs'].
- **splitting** (*{None, 'fs', 'hfs'}*) – Type of splitting. Must be one of `None`, `"fs"`, or `"hfs"`, corresponding to the inclusion of `(n,l,j)` only, `m_j`, or both `f` and `m_f` respectively.

- **expand** (*boolean, optional*) – For states with splitting, whether to return them as a list of all states. If `False`, return is a single state specification with `"all"` for the appropriate quantum numbers. If `True`, a list of all individual states is returned.

Raises

- **RydiquleError** – If `n` is not a valid atom string or integer value
- **ValueError** – If `splitting` is not one of {None, "fs", "hfs"}.

Returns

`A_QState` corresponding to the D1 excited state of the provided atom with the provided splitting, or list of `A_QState`s if `expand` is `True`.

Return type

A_QState or list of *A_QState*

Examples

The simplest use is to return the nlj quantum numbers for a particular atom's excited state of the D1 transition. Principle quantum number and string atom flags can be used interchangeably.

```
>>> atom = "Rb85"
>>> print(rq.D1_excited(atom))
(5, 1, 0.5)
>>> print(rq.D1_excited(5))
(5, 1, 0.5)
```

This function also can return states with splitting, either as a list of states or as a manifold specification.

```
>>> print(rq.D1_excited(atom, splitting="fs"))
(5, 1, 0.5, m_j='all')
>>> print(rq.D1_excited(5, splitting="fs", expand=True))
[(n=5, l=1, j=0.5, m_j=-0.5), (n=5, l=1, j=0.5, m_j=0.5)]
>>> print(rq.D1_excited(atom, splitting="hfs"))
(5, 1, 0.5, f='all', m_f='all')
```

rydiqule.atom_utils.D1_states

`rydiqule.atom_utils.`**`D1_states`**(*n: int | str, splitting: Literal[None, 'fs', 'hfs'] = None, g_splitting: Literal[None, 'fs', 'hfs'] = None, e_splitting: Literal[None, 'fs', 'hfs'] = None, expand: bool = False*) → List[*A_QState* | List[*A_QState*]]

Return the ground and excited states for the D1 line of a rydberg atom.

States are returned as a pair of `A_QStates` with the provided splitting according to the `rydberg_ground()` and *`D1_excited()`* functions with the provided splitting values passed through. When splitting is `None`, the `g_splitting` and `e_splitting` values are passed to `rydberg_ground` and `D1_excited` respectively. Otherwise, the value of `splitting` is passed to both and `g_splitting` and `e_splitting` are ignored.

By default, specifying splitting does not return a list of all states, but rather the associated specifications with `"all"` in the appropriate place. The `expand` keyword argument can be used to modify this behavior, returning a full list.

Parameters

- **n** (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must begin with ['H', 'Li', 'Na', 'K', 'Rb', 'Cs'].
- **splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of `None`, `"fs"`, or `"hfs"`, corresponding to the inclusion of `(n,l,j)` only, `m_j`, or both `f` and `m_f` respectively.
- **g_splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of `None`, `"fs"`, or `"hfs"`, corresponding to the inclusion

of (n,l,j) only, m_j, or both f and m_f respectively. Ignored if splitting is specified.

- **e_splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of None, "fs", or "hfs", corresponding to the inclusion of (n,l,j) only, m_j, or both f and m_f respectively. Ignored if splitting is specified.

Returns

Ground and D1 excited state specifications of the provided atom or principal quantum number.

Return type

list of *A_QState*

Examples

The basic use of this function is to return the A_QStates associated with the states of the D1 transition of a particular Rydberg atom. String flags and principle quantum numbers can be used interchangeably.

```
>>> atom = "Rb85"
>>> print(rq.D1_states(atom))
[(n=5, l=0, j=0.5), (n=5, l=1, j=0.5)]
>>> print(rq.D1_states(5))
[(n=5, l=0, j=0.5), (n=5, l=1, j=0.5)]
```

Furthermore, splitting can be specified either for each state individually, or just for one of the states using the optional splitting, g_splitting, or e_splitting argument.

```
>>> print(rq.D1_states(5, splitting="fs"))
[(n=5, l=0, j=0.5, m_j='all'), (n=5, l=1, j=0.5, m_j='all')]
>>> print(rq.D1_states(5, splitting="fs", expand=True))
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), (n=5, l=1, j=0.5, m_
↪j=-0.5), (n=5, l=1, j=0.5, m_j=0.5)]
>>> print(rq.D1_states(5, g_splitting="fs"))
[(n=5, l=0, j=0.5, m_j='all'), (n=5, l=1, j=0.5)]
```

rydiqule.atom_utils.D2_excited

rydiqule.atom_utils.**D2_excited**(*n: int | str, splitting: Literal[None, 'fs', 'hfs'] = None, expand: bool = False*) → *A_QState* | List[*A_QState*]

Retrieve A_QState of the excited state of the D2 line of an atom or principle quantum number.

Optionally, include fine structure splitting or hyperfine splitting, which will include all m_j or all f and m_f values respectively. By default, specifying splitting does not return a list of states, but rather the associated specification with "all" in the appropriate place. The expand keyword argument can be used to modify this behavior, returning a list.

Parameters

- **n** (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must begin with ['H', 'Li', 'Na', 'K', 'Rb', 'Cs'].
- **splitting** (*{None, 'fs', 'hfs'}*) – Type of splitting. Must be one of None, "fs", or "hfs", corresponding to the inclusion of (n,l,j) only, m_j, or both f and m_f respectively.
- **expand** (*boolean, optional*) – For states with splitting, whether to return them as a list of all states. If False, return is a single state specification with "all" for the appropriate quantum numbers. If True, a list of all individual states is returned.

Raises

- ***RydiquleError*** – If n is not a valid atom string or integer value

- **ValueError** – If splitting is not one of {None, "fs", "hfs"}.

Returns

A_QState corresponding to the D2 excited state of the provided atom with the provided splitting, or list of A_QState`s if `expand is True.

Return type

A_QState or list of A_QState

Examples

The simplest use is to return the nlj quantum numbers for a particular atom's excited state of the D2 transition. Principle quantum number and string atom flags can be used interchangeably.

```
>>> atom = "Rb85"
>>> print(rq.D2_excited(atom))
(5, 1, 1.5)
>>> print(rq.D2_excited(5))
(5, 1, 1.5)
```

This function also can return states with splitting, either as a list of states or as a manifold specification.

```
>>> print(rq.D2_excited(atom, splitting="fs"))
(5, 1, 1.5, m_j='all')
>>> print(rq.D2_excited(5, splitting="fs", expand=True))
[(n=5, l=1, j=1.5, m_j=-1.5),
 (n=5, l=1, j=1.5, m_j=-0.5),
 (n=5, l=1, j=1.5, m_j=0.5),
 (n=5, l=1, j=1.5, m_j=1.5)]
>>> print(rq.D2_excited(atom, splitting="hfs"))
(5, 1, 1.5, f='all', m_f='all')
```

rydiqule.atom_utils.D2_states

rydiqule.atom_utils.**D2_states**(*n: int | str, splitting: Literal[None, 'fs', 'hfs'] = None, g_splitting: Literal[None, 'fs', 'hfs'] = None, e_splitting: Literal[None, 'fs', 'hfs'] = None, expand: bool = False*) → List[A_QState | List[A_QState]]

Return the ground and excited states for the D1 line of a rydberg atom.

States are returned as a pair of A_QStates with the provided splitting according to the rydberg_ground() and *D2_excited()* functions with the provided splitting values passed through. When splitting is None, the g_splitting and e_splitting values are passed to rydberg_ground and D2_excited respectively. Otherwise, the value of splitting is passed to both and g_splitting and e_splitting are ignored.

By default, specifying splitting does not return a list of all states, but rather the associated specifications with "all" in the appropriate place. The expand keyword argument can be used to modify this behavior, returning a full list.

Parameters

- **n** (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must begin with ['H', 'Li', 'Na', 'K', 'Rb', 'Cs'].
- **splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of None, "fs", or "hfs", corresponding to the inclusion of (n,l,j) only, m_j, or both f and m_f respectively.
- **g_splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of None, "fs", or "hfs", corresponding to the inclusion of (n,l,j) only, m_j, or both f and m_f respectively. Ignored if splitting is specified.

- **e_splitting** (*None, "fs", "hfs, optional*) – Type of splitting for both states. Must be one of `None`, `"fs"`, or `"hfs"`, corresponding to the inclusion of `(n,l,j)` only, `m_j`, or both `f` and `m_f` respectively. Ignored if `splitting` is specified.

Returns
Ground and D2 excited state specifications of the provided atom or principal quantum number.

Return type
list of *A_QState*

Examples

The basic use of this function is to return the A_QStates associated with the states of the D2 transition of a particular Rydberg atom. String flags and principle quantum numbers can be used interchangeably.

```
>>> atom = "Rb85"
>>> print(rq.D2_states(atom))
[(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
>>> print(rq.D2_states(5))
[(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
```

Furthermore, splitting can be specified either for each state individually, or just for one of the states using the optional `splitting`, `g_splitting`, or `e_splitting` argument.

```
>>> print(rq.D1_states(5, splitting="fs"))
[(n=5, l=0, j=0.5, m_j='all'), (n=5, l=1, j=0.5, m_j='all')]
>>> print(rq.D2_states(5, splitting="fs", expand=True))
[(n=5, l=0, j=0.5, m_j=-0.5),
 (n=5, l=0, j=0.5, m_j=0.5),
 (n=5, l=1, j=1.5, m_j=-1.5),
 (n=5, l=1, j=1.5, m_j=-0.5),
 (n=5, l=1, j=1.5, m_j=0.5),
 (n=5, l=1, j=1.5, m_j=1.5)]
>>> print(rq.D1_states(5, g_splitting="fs"))
[(n=5, l=0, j=0.5, m_j='all'), (n=5, l=1, j=0.5)]
```

rydiqule.atom_utils.calc_eta

`rydiqule.atom_utils.`**`calc_eta`**(*omega: float, dipole_moment: float, beam_area: float*) → float

Calculates the eta constant needed from some experiment calculations

The value is computed with the following formula Eq. 7 of Meyer et. al. PRA 104, 043103 (2021)

$$\eta = \sqrt{\frac{\omega\mu^2}{2c\epsilon_0\hbar A}}$$

Where ω is the probing frequency, μ is the dipole moment, A is the beam area, c is the speed of light, ϵ_0 is the dielectric constant, and $\hbar$ is the reduced Plank constant.

Parameters

- **omega** (*float*) – The atomic transition frequency, in rad/s
- **dipole_moment** (*float*) – The atomic transition dipole moment, in C*m
- **beam_area** (*float*) – The cross-sectional area of the beam, in m^2

Returns
The value of eta, in root(Hz)

Return type
float

rydiqule.atom_utils.calc_kappa

rydiqule.atom_utils.**calc_kappa**(*omega: float*, *dipole_moment: float*, *density: float*) → float

Calculates the kappa constant needed for observable calculations.

The value is computed with the following formula Eq. 5 of Meyer et. al. PRA 104, 043103 (2021)

$$\kappa = \frac{\omega n \mu^2}{2c\epsilon_0\hbar}$$

Where ω is the probing frequency, μ is the dipole moment, n is atomic cloud density, c is the speed of light, ϵ_0 is the dielectric constant, and $\hbar$ is the reduced Plank constant.

Parameters

- **omega** (*float*) – Atomic transition frequency, in rad/s
- **dipole_moment** (*float*) – Dipole moment of the atomic transition, in C*m
- **density** (*float*) – The atomic number density, in m^(-3)

Returns

The value of kappa, in (rad/s)/m

Return type

float

rydiqule.atom_utils.expand_qnums

rydiqule.atom_utils.**expand_qnums**(*qstates: List[A_QState]*, *I: float | None = None*) → List[*A_QState*]

Expand all list-like A_QStates in a list.

List-like quantum numbers are defined either with a list of quantum numbers or the string "all". In the "all" case, that quantum number will be expanded into all physically allowed values of that quantum number given the preceding numbers.

Iterates through the list, expanding each A_QState specification into a list of all states matching that specification. For each state specification in the list, quantum numbers are expanded from left to right. The final list of A_QStates will respect the ordering of the initial states by ordering the states corresponding to each specification by n, l, j, m_j, f, and finally m_f

Parameters

- **qstates** (*list of A_QState*) – List of atomic quantum states specifications to be expanded.
- **I** (*Union[float,None], optional*) – Nuclear spin for the isotope of the atom. Used to calculate the f quantum number when relevant, by default None

Returns

List of all atomic states corresponding to all the specifications in the given list.

Return type

list of *A_QState*

Notes

..note::

While this function can expand arbitrary states, it should be noted that the resulting lists of states can be quite long. If they are to be used as the states of a `Cell`, these long state lists can dramatically increase computation time, and it is often worth ensuring that tracking hyperfine states individually is absolutely necessary.

Examples

A basic piece of functionality for this function is as a shorthand for all states in a given manifold.

```
>>> D1_ground = A_QState(5,0,0.5, f="all")
>>> D1_excited = A_QState(5,0,0.5,f="all")
>>> #manifolds for rubidium 87 (I=3/2)
>>> print(rq.expand_qnums([D1_ground], I=3/2))
[(n=5, l=0, j=0.5, f=1.0), (n=5, l=0, j=0.5, f=2.0)]
>>> print(rq.expand_qnums([D1_excited], I=3/2))
[(n=5, l=0, j=0.5, f=1.0), (n=5, l=0, j=0.5, f=2.0)]
>>> #manifolds for rubidium 85 (I=5/2)
>>> print(rq.expand_qnums([D1_ground], I=5/2))
[(n=5, l=0, j=0.5, f=2.0), (n=5, l=0, j=0.5, f=3.0)]
>>> print(rq.expand_qnums([D1_excited], I=5/2))
[(n=5, l=0, j=0.5, f=2.0), (n=5, l=0, j=0.5, f=3.0)]
```

Note that while this function is capable of getting large numbers of states, the resulting lists can be quite cumbersome, and be substantially slower if used in a calculation, especially for high angular momentum states.

```
>>> state = A_QState(7, 2, 2.5, f="all", m_f="all")
>>> states_all = rq.expand_qnums([state], I=7/2)
>>> print(len(states_all))
48
```

rydiqule.atom_utils.expand_single_qnum

rydiqule.atom_utils.**expand_single_qnum**(*qstate:* A_QState, *I: float* | *None* = *None*, *wildcard: str* = *'all'*) → List[*A_QState*]

Generates a list of all valid states given a particular quantum number to be expanded.

For a given `A_Qstate` spec with one or more tuple elements specified as either a list or the "all" string, returns a list of all valid state specifications matching that state specification with the first list or string element only expanded. If multiple elements of the statespec are specified with a list or string, only the first one is expanded. This function is intended as a helper function for a single quantum number, and is not designed to be used at the top-level

The the case that the element to be expanded is a list, the list returned will have a single state specification corresponding to each element of that list, and allowed quantum number rules will not be enforced. In the case that the element to be expanded is the "all" string, all valid values of that particular quantum number will be used. Note that only the `m_j`, `f`, and `m_f` quantum numbers can be expanded in this way.

Parameters

- **qstate** (`A_QState`) – NamedTuple with fields `(n, l, j, m_j, f, m_f)` representing the quantum numbers of the state. Each, element must be either a float, list of floats, or the "all" string. Only `m_j`, `f`, and `m_f` may be specified with a "all".
- **I** (*float, optional*) – The nuclear spin of the rydberg atom. Only used for calculations of valid `f` quantum numbers, defaults to 0.0

Returns

List of all possible quantum states matching the provided specification. Only the first list-like quantum number will be expanded.

Return type

List of *A_QState*

Raises

RydiquleError – If there is a string specification besides "all" in the provided state

Examples

A simple m_j expansion of the D1 states for Rubidium

```
>>> D1_g = A_QState(5,0,0.5, m_j="all")
>>> D1_e = A_QState(5,1,0.5, m_j="all")
>>> print(rq.atom_utils.expand_single_qnum(D1_g))
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5)]
>>> print(rq.atom_utils.expand_single_qnum(D1_e))
[(n=5, l=1, j=0.5, m_j=-0.5), (n=5, l=1, j=0.5, m_j=0.5)]
```

We can also expand into nuclear spin-coupled (f) states. Note that in this case we must provide the nuclear spin I to use this function on its own (in this case I=7/2 for Rb87)

```
>>> D1_g = A_QState(5,0,0.5, f="all")
>>> D1_e = A_QState(5,1,0.5, f="all")
>>> print(rq.atom_utils.expand_single_qnum(D1_g, I=7/2))
[(n=5, l=0, j=0.5, f=3.0), (n=5, l=0, j=0.5, f=4.0)]
>>> print(rq.atom_utils.expand_single_qnum(D1_e, I=7/2))
[(n=5, l=1, j=0.5, f=3.0), (n=5, l=1, j=0.5, f=4.0)]
```

While we can provide the “all” flag to expand to all states, we may want to use a specific subset of states if, for example, selection rules limit the states at play. In this case, one can pass a list for a given quantum number.

```
>>> state = A_QState(5, 2 ,2.5, m_j=[-2.5, -1.5, -0.5])
>>> print(rq.atom_utils.expand_single_qnum(state))
[(n=5, l=2, j=2.5, m_j=-2.5), (n=5, l=2, j=2.5, m_j=-1.5), (n=5, l=2, j=2.5, m_
↪j=-0.5)]
```

rydiqule.atom_utils.get_valid_f

rydiqule.atom_utils.**get_valid_f**(*state:* A_QState, *I: float* | *None* = *None*) → List[float]

Return the valid values of f for given other quantum numbers.

For a given quantum state with principal, orbital, and spin-orbit quantum numbers (n, L, J) and nuclear quantum number I, the valid values of m_f are given by

$$f = |I - J|, |I - J| + 1, ..., I + J$$

rydiqule.atom_utils.get_valid_j

rydiqule.atom_utils.**get_valid_j**(*state:* A_QState, *I: float* | *None* = *None*) → List[float]

Return the valid values of j for given other quantum numbers.

For a given quantum state with principal and orbital quantum numbers (n, l), the valid values of j are given by

$$j = |l - \frac{1}{2}|, l + \frac{1}{2}$$

Note that if both values are the same, a list of length 1 is returned.

rydiqule.atom_utils.get_valid_mf

rydiqule.atom_utils.**get_valid_mf**(*state:* A_QState, *I: float* | *None* = *None*) → List[float]

Return the valid values of m_f for given other quantum numbers.

For a given quantum state with principal, orbital, and total quantum numbers (n, L, J, f), the valid values of m_f are given by

$$m_f = -f, -f + 1, -f + 2, ..., f - 2, f - 1, f$$

rydiqule.atom_utils.get_valid_mj

`rydiqule.atom_utils.get_valid_mj(state: A_QState, I: float | None = None) → List[float]`

Return the valid values of m_J for given other quantum numbers.

For a given quantum state with principal, orbital, and total quantum numbers (n, L, J), the valid values of m_J are given by

$$m_J = -J, -J + 1, -J + 2, ..., J - 2, J - 1, J$$

rydiqule.atom_utils.ground_state

`rydiqule.atom_utils.ground_state(n: int | str, splitting: Literal[None, 'fs', 'hfs'] = None, expand: bool = False) → A_QState | List[A_QState]`

Retrieve `A_QState` for the ground state of an atom or principle quantum number.

Optionally, include fine structure splitting or hyperfine splitting, which will include all `m_j` or all `f` and `m_f` values respectively. By default, specifying splitting does not return a list of states, but rather the associated specification with `"all"` in the appropriate place. The `expand` keyword argument can be used to modify this behavior, returning a list.

Parameters

- **n** (*int or str*) – Either the string flag of the atom or the principle quantum number n of an atom. If string, must begin with ['H', 'Li', 'Na', 'K', 'Rb', 'Cs'].
- **splitting** (*{None, 'fs', 'hfs'}, optional*) – Type of splitting. Must be one of `None`, `"fs"`, or `"hfs"`, corresponding to the inclusion of `(n,l,j)` only, `m_j`, or both `f` and `m_f` respectively.
- **expand** (*boolean, optional*) – For states with splitting, whether to return them as a list of all states. If `False`, return is a single state specification with `"all"` for the appropriate quantum numbers. If `True`, a list of all individual states is returned.

Raises

- **RydiquleError** – If `n` is not a valid atom string or integer value
- **ValueError** – If `splitting` is not one of {None, "fs", "hfs"}.

Returns

`A_QState` corresponding to the ground state of the provided atom with the provided splitting, or list of `` A_QState`s if `expand `` is `True`.

Return type

A_QState

Examples

The simplest use is to return the nlj quantum numbers for a particular atom's ground state. Principle quantum number and string atom flags can be used interchangeably.

```
>>> atom = "Rb85"
>>> print(rq.ground_state(atom))
(5, 0, 0.5)
>>> print(rq.ground_state(5))
(5, 0, 0.5)
```

This function also can return states with splitting, either as a list of states or as a manifold specification.

```
>>> print(rq.ground_state(atom, splitting="fs"))
(5, 0, 0.5, m_j='all')
>>> print(rq.ground_state(5, splitting="fs", expand=True))
```

(continues on next page)

(continued from previous page)

```
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5)]
>>> print(rq.ground_state(atom, splitting="hfs"))
(5, 0, 0.5, f='all', m_f='all')
```

rydiqule.atom_utils.match_A_QState

`rydiqule.atom_utils.match_A_QState`(*qstate: A_QState*, *compare_list=[]*, *I: float | None = None*) → List[*A_QState*]

Function to return all states in a list matching the provided pattern.

States are considered a match for `qstate` if they are an element of the list returned by calling the *expand_qnums()* on `qstate`.

Parameters

- **qstate** (A_QState) – The state against which elements of the list are compared.
- **compare_list** (*list, optional*) – List of states to test. Any matching the pattern provided by `qstate` will be in the returned list, by default []
- **I** (*float, optional*) – Nuclear spin I of the atom containing the provided states, by default None

Returns

List of all the elements matching the pattern defined by `qstate`

Return type

List of *A_QState*

rydiqule.atom_utils.validate_qnums

`rydiqule.atom_utils.validate_qnums`(*qstate: A_QState*, *I: float | None = None*)

Validate that the provided named_tuple is a valid rydberg atomic state

Parameters

- **qstate** (A_QState) – Named tuple to check, should have fields `("n","l","j","m_j","f","m_f")`
- **I** (*Union[None, float], optional*) – Nuclear spin of the rydberg atom of which this is a state. If `None`, all f values are invalid automatically. Defaults to `None`

Raises

- **ValueError** – If the tuple representing the state does not have 6 elements
- **AssertionError** – If the states of the state are not physically allowed

Classes

A_QState(n, l, j[, m_j, f, m_f])	Named tuple class designed to represent the quantum numbers a state spec of an alkali atom.
QState(n, l, j[, m_j, f, m_f])	Named tuple class designed to represent the quantum numbers in the state of an alkali atom.

rydiqule.atom_utils.A_QState

class `rydiqule.atom_utils.A_QState`(*n: int*, *l: int*, *j: float*, *m_j: float | List[float] | Literal['all'] | None = None*, *f: float | List[float] | Literal['all'] | None = None*, *m_f: float | List[float] | Literal['all'] | None = None*)

Bases: NamedTuple

Named tuple class designed to represent the quantum numbers a state spec of an alkali atom. n, l, and j quantum numbers are required, with optional m_j, f, and m_f.

__init__()

Methods

__init__()	
count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

f	Alias for field number 4
j	Alias for field number 2
l	Alias for field number 1
m_f	Alias for field number 5
m_j	Alias for field number 3
n	Alias for field number 0
qnums	
stype	

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {'f': None, 'm_f': None, 'm_j': None}

_fields = ('n', 'l', 'j', 'm_j', 'f', 'm_f')

classmethod _make(*iterable*)

Make a new A_QState object from a sequence or iterable

_replace(***kwds*)

Return a new A_QState object replacing specified fields with new values

count(*value*, */ (Positional-only parameter separator (PEP 570))*)

Return number of occurrences of value.

f: float | List[float] | Literal['all'] | None

Alias for field number 4

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

j: float

Alias for field number 2

l: int

Alias for field number 1

m_f: float | List[float] | Literal['all'] | None

Alias for field number 5

m_j: float | List[float] | Literal['all'] | None

Alias for field number 3

n: int

Alias for field number 0

property qnums: Tuple[float | List[float] | Literal['all'], ...]

property stype: str

rydiqule.atom_utils.QState

class rydiqule.atom_utils.**QState**(*n: int*, *l: int*, *j: float*, *m_j: float | None = None*, *f: int | None = None*, *m_f: int | None = None*)

Bases: `NamedTuple`

Named tuple class designed to represent the quantum numbers in the state of an alkali atom. `n`, `l`, and `j` quantum numbers are required, with optional `m_j`, `f`, and `m_f`.

__init__()

Methods

__init__()	
count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

Attributes

f	Alias for field number 4
j	Alias for field number 2
l	Alias for field number 1
m_f	Alias for field number 5
m_j	Alias for field number 3
n	Alias for field number 0
qnums	Return a basic `tuple` representation of an `A_QState` with all `None` values removed.
stype	Type of state.

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {'f': None, 'm_f': None, 'm_j': None}

_fields = ('n', 'l', 'j', 'm_j', 'f', 'm_f')

classmethod _make(*iterable*)

Make a new QState object from a sequence or iterable

_replace(***kwds*)

Return a new QState object replacing specified fields with new values

count(*value*, */*)

Return number of occurrences of value.

f: int | None

Alias for field number 4

index(*value*, *start=0*, *stop=9223372036854775807*, */*)
Return first index of value.

Raises ValueError if the value is not present.

j: float
Alias for field number 2

l: int
Alias for field number 1

m_f: int | None
Alias for field number 5

m_j: float | None
Alias for field number 3

n: int
Alias for field number 0

property qnums: Tuple[float, ...]
Return a basic `tuple` representation of an `A_QState` with all `None` values removed.

Returns
Quantum numbers which are not `None`.

Return type
tuple

property stype: str
Type of state. One of "NLJ", "FS", "HFS"

Returns
String representing state type.

Return type
str

8.1.3 rydiqule.cell

Subclass of `Sensor` with functionality for representing real atoms.

Classes

`Cell`(atom_flag, atomic_states[, ...])	Subclass of `Sensor` that creates a Sensor with additional physical properties corresponding to a specific Rydberg atom.

rydiqule.cell.Cell

class rydiqule.cell.**Cell**(*atom_flag: Literal['H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs'], atomic_states: List[A_QState], cell_length: float = 0.001, gamma_transit: float | None = None, gamma_mismatch: str | dict = 'ground', beam_area: float = 1e-06, beam_diam: float | None = None, temp: float = 300.0*)

Bases: `Sensor`

Subclass of `Sensor` that creates a Sensor with additional physical properties corresponding to a specific Rydberg atom.

In addition to the core functionality of `~.Sensor`, this class requires labelling states with `namedtuple`s of quantum numbers, automatically calculating of state lifetimes and decoherences and tracking of of some physical laser parameters. A key distinction between`

a :class:`~.Cell and a *Sensor* is that a cell supports (and requires) and absolute ordering of energy between states, which allows for implicit calculation of decay rates and transition frequencies.

__init__(*atom_flag: Literal['H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs'], atomic_states: List[A_QState], cell_length: float = 0.001, gamma_transit: float | None = None, gamma_mismatch: str | dict = 'ground', beam_area: float = 1e-06, beam_diam: float | None = None, temp: float = 300.0*) → None

Initialize the Rydberg cell from the given parameters.

Parameters

- **atom_flag** (*str*) – Which atom is used in the cell for calculating physical properties with ARC Rydberg. One of {'H', 'Li6', 'Li7', 'Na', 'K39', 'K40', 'K41', 'Rb85', 'Rb87', 'Cs'}.
- **atomic_states** (*list of A_QState*) – List of *A_QState* representing the states of the atom. More details about the `A_QState` class can be found in its documentation, but it includes the elements (`n`, `l`, `j`, `m_j`, `f`, `m_f`). These represent the usual Hydrogen-like atom quantum numbers with the usual restrictions:
  - `n` must be a positive integer.
  - `l` must be a non-negative integer less than `n`.
  - `j` must be a positive half-integer such that $j = l \pm \frac{1}{2}$
  - `m_j` must be a half integer such that $-j \leq m_j \leq +j$
  - `f` must be an integer satisfying $|j - I| \leq f \leq (j + I)$
  - `m_f` must be an integer such that $-f \leq m_f \leq +f$

  Additionally, `n`,`l`, and `j` must always be specified, in addition to the following restrictions on other quantum numbers:
  - `m_j` cannot be specified with `f`
  - If `m_f` is specified, `f` must also be specified.

  All quantum numbers can be specified with lists of valid values, in which case the they will be expanded into a list of states, with one corresponding to each value. If multiple quantum numbers are specified with lists, the resulting list of states will contain all combinations of values. Furthermore, the `j`, `m_j`, `f`, and `m_f` quantum numbers can each be specified using `"all"`, which corresponds to a list of all physically allowed values for that quantum number. This convention allows quick specifications of entire manifolds of states to be added to the sensor. See the `Examples` section to see how to use these powerful specifications.
- **cell_length** (*float*) – Length of the atomic vapor in meters.
- **gamma_transit** (*float, optional*) – Decoherence due to atom transit through the optical beams. Specified in units of Mrad/s. If `None`, will calculate based on value of `beam_area`. See *add_transit_broadening()* for details on how transit broadening is treated. Default is None.
- **gamma_mismatch** (*str or dict*) – How to resolve discrepancies between calculated each state lifetime and the sum of all transition rates out of each state. In practice, these discrepancies are a result of transition pathways which exist between states when one is accounted for is `states` and one is not. For example, if there is a decoherent transition between states 3->1 and states 3->2, but `states` only includes states 0, 1, and 3, the calculated `gamma_lifetime` of state 3 will be greater than the sum of all computed `gamma_transition` values. In many cases, it is desirable to account for this decoherence in other ways. The options for handling the discrepancy are:
  - `"ground"` which adds a decoherent coupling coupling between a state `s` with a discrepancy $\Delta\gamma$ and divides $\Delta\gamma$ among all the ground states (states matching the `n,l,j` values of the lowest energy state).

  - "all" which divides $\Delta\gamma$ amongst all transitions in the *Cell* which already have a "gamma_transition" value from that state. The fraction each transition gets is weighted by the fraction of the total that that transition's "gamma_transition" value accounts for.
  - "none" which will not account for this discrepancy at all. In this case this physics is not guaranteed to be accurate and it is assumed the decoherence will be accounted for manually in other ways using *add_decoherence()*

  In all cases, the accounting is done by adding a "gamma_mismatch" value to each relevant edge, which will subsequently be accounted for when *decoherence_matrix()* is called. Note that older versions of rydiqule did not have this option and implicitly used the "ground" option, and "ground" is the current default.

- **beam_area** (*float, optional*) – Area of probing field cross-section in m^2. Used to calculate kappa and gamma_transit. Default is 1e-6.
- **beam_diam** (*float, optional*) – Diameter of the probing field cross section in meters. Used to calculate gamma_transit. If None, it is calculated from beam_area assuming the beam cross-section is a circle. Default is None.
- **temp** (*float, optional*) – Temperature of the gas in Kelvin. Used in calculations of energy level lifetime. Default is 300 K.

Raises

- ***RydiquleError*** – If at least two atomic states are not provided.
- ***AtomError*** – If atom_flag is not one of ARC's supported alkali atoms.

Warns

NLJWarning – If called using old-style state specification

Examples

All the hyperfine states for the D1 line of Rubidium-87 can be defined as follows.

```
>>> from rydiqule import A_QState
>>> D1_g = A_QState(5, 0, 0.5, f="all", m_f="all")
>>> D1_e = A_QState(5, 1, 0.5, f="all",m_f="all")
>>> c = rq.Cell("Rb87",[D1_e, D1_g])
>>> for state in c.states:
...     print(state)
(5, 1, 0.5, f=1.0, m_f=-1.0)
(5, 1, 0.5, f=1.0, m_f=0.0)
(5, 1, 0.5, f=1.0, m_f=1.0)
(5, 1, 0.5, f=2.0, m_f=-2.0)
(5, 1, 0.5, f=2.0, m_f=-1.0)
(5, 1, 0.5, f=2.0, m_f=0.0)
(5, 1, 0.5, f=2.0, m_f=1.0)
(5, 1, 0.5, f=2.0, m_f=2.0)
(5, 0, 0.5, f=1.0, m_f=-1.0)
(5, 0, 0.5, f=1.0, m_f=0.0)
(5, 0, 0.5, f=1.0, m_f=1.0)
(5, 0, 0.5, f=2.0, m_f=-2.0)
(5, 0, 0.5, f=2.0, m_f=-1.0)
(5, 0, 0.5, f=2.0, m_f=0.0)
(5, 0, 0.5, f=2.0, m_f=1.0)
(5, 0, 0.5, f=2.0, m_f=2.0)
```

Methods

`__init__`(atom_flag, atomic_states[, ...])	Initialize the Rydberg cell from the given parameters.
`add_coupling`(states, **kwargs)	Add a coupling between states or groups of states.
`add_coupling_group`(states1, states2, label)	Adds a group of couplings to a Sensor.
`add_couplings`(*couplings, **extra_kwargs)	Add any number of couplings between pairs of states.
`add_decoherence`(statespecs, gamma, **kwargs)	Add a coupling between states or groups of states.
`add_decoherence_group`(states1, states2, ...)	Adds a group of decoherences to the Sensor.
`add_energy_shift`(statespec, shift, **kwargs)	Add an energy shift to a single state or a group of states.
`add_energy_shift_group`(states, shift[, ...])	Add energy shifts to a group of states, optionally with a modifying prefactor for each.
`add_energy_shifts`(shifts)	Wrapper for `Sensor.add_energy shift b()`.
`add_self_broadening`(state, gamma[, label, ...])	Specify self-broadening (such as collisional broadening) of a level.
`add_self_broadening_group`(states, gamma[, ...])	Specify self-broadening (such as collisional broadening) of a group of states.
`add_single_coupling`(states[, ...])	Overload of `add_single_coupling()`, which allows for alternate specifications and automatic calculations of some parameter.
`add_single_decoherence`(states, gamma[, ...])	Add decoherent coupling to the graph between two states.
`add_single_energy_shift`(state, shift[, label])	Add an energy shift to a state.
`add_transit_broadening`(gamma_transit[, ...])	Overload of the `add_transit_broadening()` method of `Sensor` which automatically populates the `repop` dictionary with a equal decay to all sublevels of the ground `(n, l, j)` state.
`axis_labels`()	Get a list of axis labels for stacked hamiltonians.
`coupling_subgraph`(coupling)	Returns a subgraph view of the couplings graph corresponding to `coupling`.
`couplings_with`(*keys[, method])	Returns a version of self.couplings with only the keys specified.
`decoherence_matrix`()	Build a decoherence matrix out of the decoherence terms of the graph.
`dm_basis`()	Generate basis labels of density matrix components.
`get_couplings`()	Returns the couplings of the system as a dictionary
`get_doppler_shifts`()	Returns the Hamiltonian with only detunings set to the most probable doppler shift values for each spatial dimension.
`get_hamiltonian`()	Creates the Hamiltonians from the couplings defined by the fields.
`get_hamiltonian_diagonal`(values[, no_stack])	Apply addition and subtraction logic corresponding to the direction of the couplings.
`get_parameter_mesh`()	Returns the parameter mesh of the sensor.
`get_rotating_frames`()	Determines the rotating frames for the disconnected subgraphs.
`get_time_dependence`()	Function which returns a list of the `time_dependence` functions.
`get_time_hamiltonian`(t)	Get the system hamiltonian at a specific time, t.
`get_time_hamiltonian_components`()	Get time-dependent components of the hamiltonian.
`get_transition_frequencies`()	Gets an array of the diagonal elements of the Hamiltonian from the field detunings.
`get_value_dictionary`(key)	Get subset of dictionary coupling parameters.
`group_variable_parameters`([apply_mesh])	

continues on next page

Table 8.14 – continued from previous page

int_states_map([invert])	Get a dictionary mapping between state labels and their corresponding integer ordering.
level_ordering()	Return a list of the states in the `Cell` in ascending energy order.
set_experiment_values(probe_freq, kappa[, ...])	`Sensor` specific method.
set_gamma_matrix(gamma_matrix)	Set the decoherence matrix for the system.
spatial_dim()	Returns the number of spatial dimensions doppler averaging will occur over.
states_with_spec(statespec)	Return a list of all states in the sensor matching the `state_spec` pattern.
unzip_parameters(zip_label[, verbose])	Remove a set of zipped parameters from the internal zip_labels list.
variable_parameter_sort(par)	Assistance function which determines the sorting order of elements parameters in sensor.
variable_parameters([apply_mesh])	Property to retrieve the values of parameters that were stored on the graph as arrays.
zip_parameters(parameters[, zip_label])	Define 2 scannable parameters as "zipped" so they are scanned in parallel.
zip_zips(*zip_labels[, new_label])	Combine multiple parameter zips into a single zip.

Attributes

atom_mass	Mass of an atom in the vapor cell, in kilograms.
basis_size	Property to return the number of nodes on the Sensor graph.
beam_area	Cross-sectional area of the probing beam, in square meters.
cell_length	Optical path length of the medium, in meters.
eta	Get the eta value for the system.
kappa	Property to calculate the kappa value of the system.
probe_freq	Get the probe transition frequency, in rad/s.
probe_tuple	Coupling edge that corresponds to the probing field.
states	Property which gets a list of labels for the sensor in the order defined in *__init__()*.
temp	Temperature of the vapor cell, in Kelvin.
vP	Most probable speed of the 3D Maxwell-Boltzmann distribution.
v_th	Thermal velocity of the atoms in vapor cell, in meters per second.

_add_coherent_data(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], **field_params*) → None

Function for internal use which will ensure the supplied couplings is valid, add the field to self.couplings.

Exists to abstract away some of the internally necessary bookkeeping functionality from user-facing classes.

Parameters

- **states** (*tuple*) – The integer pair of states to be coupled.
- ****field_params** (*dict*) – The dictionary of couplings parameters. For details on the keys of the dictionary see *add_coupling()*.

_add_decoherence_rates()

Helper function to add natural decay rates to all transitions in the cell. Values for decay rates are calculated

with arc, and skipped if selection rules prohibit the transition or if the second state is a higher energy than the first state

_add_gamma_mismatch_to_all(*state: A_QState*)

Helper function which implements the "all" option of `_add_gamma_mismatches()` for a single state `state`.

_add_gamma_mismatch_to_ground(*state: A_QState*)

Helper function which implements the "ground" option of `_add_gamma_mismatches()` for a single state `state`.

_add_gamma_mismatches(*method: str | dict = 'ground'*)

Adds couplings to the graph accounting for differences between computed lifetimes and decay rates.

In a cell in which all atomic states are accounted for, the computed values of `gamma_lifetime` for a particular state will be equal to the sum of all `gamma_transition` values on edges leaving that state. However, it is not always desirable to account for all states in this way for simplicity or computational complexity reasons. This function allows the `Cell` to account for any differences in these values that arise as a result of excluding physical states from a `Cell`. There are multiple ways to resolve these discrepancies, specified by the `method` argument, which is detailed in the `Parameters` section.

Parameters

method (*str or dict mapping states to str*) – The method by which discrepancies in computed values are resolved. The available methods are as follows:

- `"ground"` which adds a decoherent coupling coupling between a state `s` with a discrepancy $\Delta\gamma$ and divides $\Delta\gamma$ among all the ground states (states matching the `n`, `l`, `j` values of the lowest energy state).
- `"all"` which divides $\Delta\gamma$ amongst all states in the `Cell` which already have a `"gamma_transition"` value. The fraction each transition gets is weighted by the fraction of the total that transitions `"gamma_transition"` value accounts for. If this method is used, every state in the `Cell` must have at least one dipole-allowed decay path.
- `"none"` which will not account for this discrepancy at all. In this case this physics is not guaranteed to be accurate and it is assumed the decoherence will be accounted for manually in other ways using `"~.add_decoherence"()`

Note that in addition to one of these strings `method` can be a dictionary which maps states to one of these strings. In this case, the discrepancy is resolved separately for each state using the method specified for that state. Defaults to "ground"

Raises

- **ValueError** – If the method is not one of the allowed strings or a dictionary mapping states to one of the allowed strings.
- ***RydiquleError*** – If the "all" option is selected for `method` and any of the states have no lower state to decay to.

_add_state_energies()

Helper function to add all the "energy" key to all the nodes of the graph for the state energies relative to the first state.

All energies are relative to the ground state, defined as the $nS^{1/2}$ state, where n is the principle quantum number of the lowest energy atomic state.

_add_state_lifetimes()

Helper function to add all "gamma_lifetime" key-value appropriate to the atom to all nodes on the graph.

_coupling_with_label(*label: str | Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]*) → Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]

Helper function to return the pair of states corresponding to a particular label string. For internal use.

_expand_dims()

Converts the 1-D arrays in the sensor into shapes that allows for rydiqule stacking.

_probe_tuple: **StateSpecs** | **None** = **None**

_remove_edge_data(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], kind: str*)

Helper function to remove all data that was added with a `add_coupling()` call or `add_decoherence()` call. Needed to ensure that two nodes do not have coherent couplings pointing both ways and to invalidate existing zip parameter couplings.

Parameters

- **states** (*tuple*) – Edge from which to remove data.
- **kind** (*str*) – What type of data to remove. Valid options are `coherent` coherent couplings or the incoherent key to be cleared (must start with `gamma`).

Raises

RydiquleError – If `kind` is not `'coherent'` and doesn't begin with `'gamma'`

_stack_shape(*time_dependence: Literal['steady', 'time', 'all'] = 'all'*) → Tuple[int, ...]

Internal function to get the shape of the tuple preceding the two hamiltonian axes in `get_hamiltonian()`

_states_valid(*states: Sequence*) → Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]

Confirms that the provided states are in a valid format.

Typically used internally to validate states added. If provided as a form other than a tuple, first casts to a tuple for consistent indexing.

Checks that `states` contains 2 elements, can be interpreted as a tuple, and that both states lie inside the basis.

Parameters

states (*iterable*) – iterable of to validate. Should be a pair of integers that can be cast to a tuple.

Returns

Length 2 tuple of validated state labels.

Return type

tuple

Raises

- ***RydiquleError*** – If `states` has more than two elements.
- **TypeError** – If `states` cannot be converted to a tuple.
- ***RydiquleError*** – If either state in `states` is outside the basis.

_vP: **float** | **None** = **None**

_validate_input_states(*atomic_states: List[A_QState]*)

Helper function to check that input states are compatible and defined

add_coupling(*states: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], **kwargs*)

Add a coupling between states or groups of states.

Wraps the `add_single_coupling()` and `add_coupling_group()` functions, and dispatches to the appropriate one depending on the number of states in the `states` argument. Additional keyword arguments will be passed unmodified to the relevant method. See documentation of those functions for details on keyword argument options.

If each state specification in `states` correspond to a single state, the corresponding states will be passed to *add_single_coupling()*. If either or both specifications correspond to multiple states, the corresponding lists will be passed as the `states1` and `states2` lists in *add_coupling_group()*.

If this is the first time `add_coupling` has been called for this `Sensor`, sets the `probe_tuple` attribute to the `states` specification , which is used as the default, for calculating observable values, in a *Solution* after solving. For this reason, this function is preferred over *add_single_coupling()* and *add_coupling_group()* outside of special circumstances. If couplings are added with either of the specific dispatched functions, `probe_tuple` should be set manually.

Parameters

- **states** (*tuple of Statespecs*) – The states or state manifolds of the coupling. If both are integers or state specifications matching a single state in the `Sensor`, *add_single_coupling()* is dispatched. If either argument is a string pattern matching multiple states, *add_coupling_group()* is dispatched.
- ****kwargs** – Additional keyword arguments passed to the relevant function. See the documentation for *add_single_coupling()* and *add_coupling_group()* for details on valid keyword arguments.

Notes

..note:

Outside of specific use cases for users well-versed in the `rydiqule` code base, this method is preferred over *add_single_coupling()* and *add_coupling_group()* since it appropriately handles necessary backend bookkeeping.

Examples

Couplings are added identically regardless of how states are labelled.

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

```
>>> s = rq.Sensor(['g','e'])
>>> s.add_coupling(('g','e'), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

Couplings can have list-like parameters, in which case the resulting rydiqule will compute hamiltonians for all values simultaneously. Here 101 x 21 = 2,121 2x2 Hamiltonians are generated simultaneously, with one for every combination of parameters, and arranged into a single array.

```
>>> s = rq.Sensor(2)
>>> det=np.linspace(-10, 10, 101)
>>> rabi = np.linspace(-1, 1, 21)
>>> s.add_coupling((0,1), detuning=det, rabi_frequency=rabi, label="laser")
>>> print(s.get_hamiltonian().shape)
(101, 21, 2, 2)
```

Couplings can be be defined between manifolds of states with state specifications. The values for rabi frequencies of individual states are modified by the `coupling_coefficients` keyword argument. To avoid cumbersome numbers of nested brackets, it is advisable to name manifolds with variables. Note that `` StateSpec`s can be expanded with :func:`~.sensor_utils.expand_statespec `` for this purpose.

```
>>> g = (0,0) #statespec for ground
>>> excited = (1,[-1,0,1]) #statespec for excited
>>> [e1,e2,e3] = rq.sensor_utils.expand_statespec(excited)
>>> cc = {
...     (g, e1): 0.25,
...     (g, e2): 0.5,
...     (g, e3): 0.25,
... } # coupling coefficients
>>> s = rq.Sensor([g, excited])
>>> s.add_coupling((g, excited), rabi_frequency=10, detuning=1, coupling_
↪coefficients=cc, label="laser")
>>> print(s.get_hamiltonian())
[[ 0.   +0.j  1.25+0.j  2.5 +0.j  1.25+0.j]
 [ 1.25-0.j -1.   +0.j  0.   +0.j  0.   +0.j]
 [ 2.5 -0.j  0.   +0.j -1.   +0.j  0.   +0.j]
 [ 1.25-0.j  0.   +0.j  0.   +0.j -1.   +0.j]]
```

This function sets the `probe_tuple` for the first call, but not subsequent calls. This makes it preferred over *add_single_coupling()* and *add_coupling_group()*, which do not have this behavior.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> e2 = (2,0)
>>> s = rq.Sensor([g, e1, e2])
>>> print(s.probe_tuple)
None
>>> s.add_coupling((g, e1), rabi_frequency=10, detuning=1, label="red")
>>> print(s.probe_tuple)
((0, 0), (1, [-1, 0, 1]))
>>> s.add_coupling((e1,e2), rabi_frequency=1, detuning=2, label="blue")
>>> print(s.probe_tuple)
((0, 0), (1, [-1, 0, 1]))
```

For state manifolds, list-like parameters are automatically zipped. See `Sensor.zip_parameters()` for more details on the mechanics of zipping parameters.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> s = rq.Sensor([g, e1])
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((g, e1), rabi_frequency=10, detuning=det, label="red")
>>> print(s.couplings.edges)
[((0, 0), (1, -1)), ((0, 0), (1, 0)), ((0, 0), (1, 1))]
>>> print(s._zip_labels)
['red_detuning']
>>> print(s.get_hamiltonian().shape)
(11, 4, 4)
```

add_coupling_group(*states1: List[int | str | Tuple[float, ...]], states2: List[int | str | Tuple[float, ...]], label: str, rabi_frequency: float | List[float] | ndarray | None = None, detuning: float | List[float] | ndarray | None = None, transition_frequency: float | None = None, coupling_coefficients: dict | None = None, time_dependence: Callable | Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Callable | None] | None = None, **kwargs*)

Adds a group of couplings to a Sensor.

Given 2 lists of states, iterates over each combination of states in the two lists, add performs the *add_single_coupling()* on that pair of states. All additional parameters are passed directly to the `add_sin-`

gle_coupling function.

Additionally, a multiplicative factor can be applied to the rabi frequency of each coupling (i.e. Clebsch-Gordon coefficients). These factors are provided by the `` cc` (coupling coefficient) parameter as a dictionary with keys corresponding to state pairs in the groups, and the value being the multiplicative factor applied to `rabi_frequency `` when the Hamiltonian is generated. Note that these `cc` values cannot be arrays. The corollary for state energy (e.g. for `detuning` or `transition_frequency`) is handled via `add_energy_shifts()`. If no dictionary is supplied for for `coherent_cc`, *all* coupling coefficients are set to 1.0, effectively meaning that the base `rabi_frequency` supplied is what is added to the Hamiltonian. If a dictionary is supplied, any couplings whose coefficient is not specified by the dictionary will be left off the graph.

If any of the parameters are specified as arrays, the associated couplings will be applied to all couplings and automatically zipped together with the label specified by `label`. For the purposes of axis labelling, the parameters will be zipped in the order`rabi_frequency`, `detuning`, `transition_frequency`.

Note that unlike `add_coupling()`, this function does not set the `probe_tuple` attribute, so if used to add the first coupling, `probe_tuple` must be set manually.

Parameters

- **states1** (*list[str or int]*) – List of states in the lower energy group of states. Must be integers or string values which correspond to states in the Sensor.
- **states2** (*list[str or int]*) – List of states in the higher energy group of states. Must be integers or string values which correspond to states in the Sensor.
- **label** (*str*) – Required string label denoting what the group of couplings is called. Used to apply a label in `zip_parameters()`.
- **rabi_frequency** (*ScannableParameter, optional*) – Floating point value or list of values for the base rabi frequency of the coupling group. Multiplied by values specified in `cc` for individual couplings, often accounting for variations in dipole moment. Default is None.
- **detuning** (*ScannableParameter, optional*) – Base detuning for the coupling group. If specified, every coupling in the group will be treated in the rotating frame. Can be modified through energy level shifts on individual states specified by `add_energy_shift()`. Default is None.
- **transition_frequency** (*float, optional*) – Base transition frequency for the coupling group. Individual states can be shifted via `add_energy_shift()`. Default is None.
- **coupling_coefficients** (*dict, optional*) – Individual coupling coefficients passed to the `add_single_coupling()` method. If provided, defined by a dictionary keyed with tuples of states corresponding to couplings in this group, with values equal to the coupling coefficient to be passed to the `add_single_coupling` call for that coupling. If any entries are absent in the provided dictionary, they are assumed to not be coupled, and no coupling will be added for that transition. If `None`, defaults to a dictionary containing every coupling in coupling in the group with `None` for all values (defaulting to 1.0 when passed to `add_single coupling`). Defaults to `None`.
- **time_dependence** (*scalar function or dict of scalar functions, optional*) – Time-dependent scalar factor that is multiplied by the rabi frequency in Hamiltonian generation. Can be specified as a single function, in which case the function will be used as the `time_dependence` argument for each coupling in the group (see `add_single_coupling()`), Can also be specified as a dictionary mapping state pairs in the coupling to individual functions which will be applied to the associated coupling in the same manner. In the case of a dictionary specification, each unspecified coupling will default to `time_dependence=None`.

Raises

RydiquleError – If `states1` and `states2` only have one state. Use `add_coupling()` instead.

> **Note**
>
> > **Note**
> >
> > The `add_coupling()` is typically preferred over this method, since it allows for shorthand specification of groups, and sets the `Sensor.probe_tuple` attribute.
>
> > **Note**
> >
> > If a `CouplingNotAllowedError` is raised while adding the individual couplings for the group, couplings that raised the error will be ignored.

Examples

Energy shifts added to remove degenerate energy levels. If no clebsch-gordon coefficients are supplied, ALL default to 1

```
>>> s = rq.Sensor(['a1', 'a2', 'b1', 'b2'])
>>> s.add_energy_shifts({'a2':0.1, 'b2':0.1})
>>> s.add_coupling_group(['a1','a2'], ['b1','b2'], detuning=1, rabi_
↪frequency=1, label='example')
>>> s.get_hamiltonian()
array([[ 0. +0.j,  0. +0.j,  0.5+0.j,  0.5+0.j],
       [ 0. +0.j,  0.1+0.j,  0.5+0.j,  0.5+0.j],
       [ 0.5-0.j,  0.5-0.j, -1. +0.j,  0. +0.j],
       [ 0.5-0.j,  0.5-0.j,  0. +0.j, -0.9+0.j]])
```

If the cc dictionary is specified, any unspecified terms are skipped on the graph. Note that although `(0,3)` is in the coupling group, it is omitted from the graph since it is not in `coupling_coefficients`.

```
>>> s = rq.Sensor(4)
>>> cc = {(0,1):0.5, (0,2):0.5}
>>> s.add_coupling_group([0],[1,2,3], detuning=1, rabi_frequency=1,␣
↪coupling_coefficients=cc, label='foo')
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 4 states and 2 coherent␣
↪couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_0, coherent_cc: 0.5}
    (0,2): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_1, coherent_cc: 0.5}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

For list-like parameters, the couplings are treated as originating from a single laser and that parameter is zipped across all couplings in the group.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> s = rq.Sensor([g, e1])
>>> det = np.linspace(-1,1,11)
```

(continues on next page)

(continued from previous page)

```
>>> s.add_coupling_group([(0,0)], [(1,-1), (1,0), (1,1)],
...                       rabi_frequency=10, detuning=det, label="red")
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent␣
↪couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 10, detuning: <parameter with 11␣
↪values>, phase: 0, kvec: (0, 0, 0), label: red_0, coherent_cc: 1.0, red_
↪detuning: detuning}
    ((0, 0),(1, 0)): {rabi_frequency: 10, detuning: <parameter with 11␣
↪values>, phase: 0, kvec: (0, 0, 0), label: red_1, coherent_cc: 1.0, red_
↪detuning: detuning}
    ((0, 0),(1, 1)): {rabi_frequency: 10, detuning: <parameter with 11␣
↪values>, phase: 0, kvec: (0, 0, 0), label: red_2, coherent_cc: 1.0, red_
↪detuning: detuning}
Decoherent Couplings:
    None
Energy Shifts:
    None
Zip Labels:
    ['red_detuning']
>>> print(s.get_hamiltonian().shape)
(11, 4, 4)
```

add_couplings(**couplings: Dict*, ***extra_kwargs*) → None

Add any number of couplings between pairs of states.

Acts as an alternative to calling `add_coupling()` individually for each pair of states. Can be used interchangeably up to preference, and all of keyword `add_coupling()` are supported dictionary keys for dictionaries passed to this function.

Note that since this function wraps `add_coupling()`, the first element of `couplings` will be used to set `probe_tuple`.

Parameters

- **couplings** (*tuple of dicts*) – Any number of dictionaries, each specifying the parameters of a single field coupling 2 states. For more details on the keys of each dictionary see the arguments for `add_coupling()`. Equivalent to passing each dictionaries keys and values to `add_coupling()` individually.
- ****extra_kwargs** (*dict*) – Additional keyword-only arguments to pass to the relevant `add_coupling` method. The same arguments will be passed to each call of `add_coupling()`. Often used for warning suppression. Can also be used to define a common coupling parameter for each coupling.

Examples

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 3 states and 2 coherent␣
↪couplings.
States: [0, 1, 2]
Coherent Couplings:
```

(continues on next page)

(continued from previous page)

```
    (0,1): {rabi_frequency: 1, detuning: 2, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 3, detuning: 4, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: (1,2)}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

add_decoherence(*statespecs: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], gamma: float | List[float] | ndarray, **kwargs*)

Add a coupling between states or groups of states.

Wraps the `add_single_decoherence()` and `add_decoherence_group()` functions, and dispatches to the appropriate one depending on the formatting of the `states` argument. Additional keyword arguments will be passed unmodified to the relevant method. See documentation of those functions for details on keyword argument options.

Parameters

- **states** (`tuple of StateSpec`) – The states or state manifolds of the decoherent coupling. If both are integers or string patterns matching a single state in the `Sensor`, `add_single_decoherence()` is dispatched. If either argument is a specification matching multiple states, `add_decoherence_group()` is dispatched.
- **gamma** (`float or Sequence`) – The decoherence rate, in Mrad/s.
- ****kwargs** – Additional keyword arguments passed to the appropriate function. See documentation for `add_single_decoherence()` and `add_decoherence_group()` for more details on valid keyword arguments.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), 0.1, label="misc")
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.  0. ]
 [0.1 0.  0. ]]
```

To add multiple decoherence effects to the same term, use a different label for each.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), 0.1, label='foo')
>>> s.add_decoherence((2,0), 0.15, label='bar')
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.   0.   0.  ]
 [0.25 0.   0.  ]]
```

Just like coherent coupling parameters, decoherence values can be passed as list-like objects and scanned. This adjusts the hamiltonian shape for clear broadcasting.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

Upper and lower states can also be regex strings matched against states in the Sensor just like for coherent couplings.

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> s.add_coupling(states=('g','e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=('e1','e2'), detuning=1, rabi_frequency=1)
>>> gamma = np.linspace(0, 0.3, 3)
>>> cc = {('e1','g'):0.25, ('e2','g'):0.75}
>>> s.add_decoherence((['e1','e2'],'g'), gamma, label="test", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[[0.     0.     0.    ]
  [0.     0.     0.    ]
  [0.     0.     0.    ]]

 [[0.     0.     0.    ]
  [0.0375 0.     0.    ]
  [0.1125 0.     0.    ]]

 [[0.     0.     0.    ]
  [0.075  0.     0.    ]
  [0.225  0.     0.    ]]]
```

Also just like coherent couplings, decoherent coupling can be defined over manifolds using state specifications. The interface is identical.

```
>>> g = (0,[-1,1])
>>> e = (1,[-1,1])
>>> cc = {
...     ((1,-1),(0,-1)):1,
...     ((1,1),(0,-1)):2,
...     ((1,-1),(0,1)):1,
...     ((1,1),(0,1)):2,
... }
>>> s = rq.Sensor([g,e])
>>> s.add_coupling((g,e), detuning=1, rabi_frequency=1, label='foo')
>>> s.add_decoherence((e,g), 0.1, coupling_coefficients=cc, label='bar')
>>> print(s.decoherence_matrix())
[[0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.1 0.1 0.  0. ]
 [0.2 0.2 0.  0. ]]
```

add_decoherence_group(*states1: List[int | str | Tuple[float, ...]], states2: List[int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, label: str, coupling_coefficients: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], float] | None = None*)

Adds a group of decoherences to the Sensor.

Given 2 lists of states, adds a single coupling across each combination of states between the first and second lists. Then, if gamma is a array-like of values, automatically performs `zip_parameters()` on all decoherences added as part of this function so they share an axis when `decoherence_matrix()` is called.

Scaling multiplicative factors for `gamma` must be applied per pair of states using `decoherent_cc`, a dictionary of coefficients determining coupling strengths. If a pair is not in `decoherent_cc`, it is assumed to have a coupling coefficient of zero, and will be omitted from the graph. If `decoherent_cc` is `None`, all couplings are assumed to have a relative strength of 1.

Parameters

- **states1** (*List of State*) – The list of states out of which population is decaying. Each element of the list must be a state in this `Sensor`.
- **states2** (*List of State*) – The list of states into which population is decaying. Each element of the list must be a state in this `Sensor`.
- **label** (*str*) – Required string label denoting what the group of dephasings is called. Used to apply a label to the zip.
- **gamma** (*ScannableParameter*) – Base decoherence rate between the two groups of states, in units of Mrad/s. Multiplied by the corresponding values in the `decoherent_coupling` dictionary.
- **coupling_coefficients** (*dict, optional*) – Coefficients describing the relative coupling strengths for decoherences in the group. Treated as modifications to the "base" dephasing rate specified by the `gamma` argument. The gamma of individual decoherences will be the `gamma` argument multiplied by the corresponding value in this dictionary. If `None`, all couplings in the group are assumed to have a coefficient of 1.0 if specified, all unspecified coupling pairs are ignored.

Raises

- **ValueError** – If the either of the states strings provided cannot be parsed as a regex pattern
- **ValueError** – If `states1` and `states2` only have one state. Use `add_decoherence()` instead.

Examples

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> s.add_coupling(states=('g','e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=('e1','e2'), detuning=1, rabi_frequency=1)
>>> cc = {('e1','g'):0.25, ('e2','g'):0.75}
>>> s.add_decoherence_group(['e1','e2'],['g'], 0.1, "test", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[0.    0.    0.   ]
 [0.025 0.    0.   ]
 [0.075 0.    0.   ]]
```

Unlike `Sensor.add_decoherence()`, this function does not accept state specifications. Upper and lower states must be passed as lists. As this tends to be a little clunkier, `Sensor.add_decoherence()` is usually preferred.

```
>>> g = (0,[-1,1])
>>> e = (1, [-1,1])
>>> list_g = [(0, -1), (0, 1)]
>>> list_e = [(1, -1), (1, 1)]
>>> cc = {
```

(continues on next page)

(continued from previous page)

```
...     ((1,1),(0,1)): 0.4,
...     ((1,1),(0,-1)): 0.1,
...     ((1,-1),(0,1)): 0.1,
...     ((1,-1),(0,-1)): 0.4
... }
>>> s = rq.Sensor([g,e])
>>> print(s.states)
[(0, -1), (0, 1), (1, -1), (1, 1)]
>>> s.add_decoherence_group(list_e, list_g, 0.1, "foo", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[0.   0.   0.   0.  ]
 [0.   0.   0.   0.  ]
 [0.04 0.01 0.   0.  ]
 [0.01 0.04 0.   0.  ]]
```

add_energy_shift(*statespec: int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], shift: float | List[float] | ndarray, **kwargs*)

Add an energy shift to a single state or a group of states.

`statespec` can be provided either as a single state in the Sensor or as a valid state specification matching a group of states. When `statespec` matches a single state, `add_single_energy_shift()` method will be dispatched. In the case of a multi-state specification, the `add_energy_shift_group()` method will be dispatched applying an individual shift to all states with labels matching the specification provided.

Note that an energy shifts are applied to the underlying graph as a self-edge connecting a node to its self, not as data on the node its self.

Additional arguments for either dispatched function are passed normally via `**kwargs`

Parameters

- **state_spec** (*StateSpec*) – Integer or string label matching a state in the `Sensor`, or state specification matching one or more states in the `Sensor`. The number of states this corresponds to will affect which internal function is dispatched.
- **shift** (*float or array-like*) – The energy shift to apply to the matching state or states in Mrad/s. Note that if it corresponds to multiple states, the `prefactors` argument of `add_energy_shift_group()` will be multiplied by this value for the corresponding state.

Raises

RydiquleError – If the state provided does not match any state in the `Sensor`.

Examples

The basic use of `add_energy_shift` is to add terms to the diagonal of the hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_energy_shift(1, 1)
>>> s.add_energy_shift(2, 2.5)
>>> print(s.couplings.edges(data=True))
[(1, 1, {'e_shift': 1, 'label': '1'}), (2, 2, {'e_shift': 2.5, 'label': '2
↪'})]
>>> print(s.get_hamiltonian())
[[0. +0.j 0. +0.j 0. +0.j]
 [0. +0.j 1. +0.j 0. +0.j]
 [0. +0.j 0. +0.j 2.5+0.j]]
```

`add_energy_shift` can be used with state specifications.

```
>>> s = rq.Sensor([(0,0), (1,[-1,0,1])])
>>> prefactors = {(1,i):i for i in [-1,0,1]}
>>> s.add_energy_shift((1, [-1,0,1]), 0.1, prefactors=prefactors)
>>> print(s.couplings.edges(data="e_shift"))
[((1, -1), (1, -1), -0.1), ((1, 0), (1, 0), 0.0), ((1, 1), (1, 1), 0.1)]
>>> print(s.get_hamiltonian())
[[ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j -0.1+0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0.1+0.j]]
```

add_energy_shift_group(*states: List[int | str | Tuple[float, ...]], shift: float | List[float] | ndarray, prefactors: dict | None = None, zip_label: str | None = None*)

Add energy shifts to a group of states, optionally with a modifying prefactor for each.

Given a list of states, calls `add_single_energy_shift()` on each one with the provided energy shift. Shifts are modified by a multiplicative factor defined by the `prefactors` dictionary. The dictionary is keyed with states that are elements of `states` with entries corresponding to a factor multiplied by the base `shift` argument for each state. When energy shifts are array-like, the `e_shift` attribute corresponding to each self-edge will be zipped with `zip_parameters()`.

Parameters

- **states** (*list of states*) – List of states to include in the group.
- **shift** (*float or array-like*) – The base value of the energy shift to apply the states. Will be modified by entries of the `prefactors` dictionary.
- **prefactors** (dict or `None`, optional) – Dictionary of values by which to multiply the base `shift` parameter for each each state. Keys are elements of the `states` list, entries are the corresponding factor by which to multiply `shift` for that state. If `None`, all prefactors are set to 1. If not `None`, the prefactors for any non-specified values will be set to zero. Default is `None`.
- **zip_label** (str or `None`, optional) – Label passed to `zip_parameters()` when the shift is provided as an array-like when all states in the group are zipped together. Defaults to `None`.

Raises

RydiquleError – If the supplied energy shift is not a float and cannot be interpreted as a numpy

Examples

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> factors = {'e1':1, 'e2':2}
>>> s.add_energy_shift_group(["e1","e2"], 0.1, prefactors=factors)
>>> print(s.couplings.edges(data='e_shift'))
[('e1', 'e1', 0.1), ('e2', 'e2', 0.2)]
>>> print(s.get_hamiltonian())
[[0. +0.j 0. +0.j 0. +0.j]
 [0. +0.j 0.1+0.j 0. +0.j]
 [0. +0.j 0. +0.j 0.2+0.j]]
```

add_energy_shifts(*shifts: dict*)

Wrapper for `Sensor.add_energy shift b()`.

Shifts are specified with the `shifts` dictionary, which is keyed with states and has values corresponding to the energy shift applied to the state in Mrad/s. Error handling and validation is done with the `add_energy_shift()` function.

Parameters

shifts (*dict*) – Dictionary keyed with states with values corresponding to the energy shift, in Mrad/s, of the corresponding state.

add_self_broadening(*state: int | str | Tuple[float, ...], gamma: float | List[float] | ndarray, label: str = 'self', decoherent_cc: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], float] | None = None*)

Specify self-broadening (such as collisional broadening) of a level.

Equivalent to calling *add_decoherence()* and specifying both states to be the same, with the "self" label. For more complicated systems, it may be useful to further specify the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Parameters

- **state** (*State*) – State or states to which the broadening will be added. Using a regular expression allows for specifying self broadening of a group of states. In this case, `mult-factor` is used to define relative amplitudes.
- **gamma** (*float or sequence*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. By default, decay will be stored on the graph edge as `"gamma_self"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`
- **mult-factor** (*dict*) – Dictionary mapping of the scaling factors to apply to the self broadening of each state in a group specified via regular expression.

Notes

Note

Just as with the *add_decoherence()* function, adding a decoherence value with a label that already exists will overwrite an existing decoherent transition with that label. The "self" label is applied to this function automatically to help avoid an unwanted overwrite.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_self_broadening(1, 0.1)
>>> print(s.couplings.edges(data=True))
[(1, 1, {'gamma_self': 0.1, 'label': '(1,1)'})]
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.1 0. ]
 [0.  0.  0. ]]
```

add_self_broadening_group(*states: List[int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, label: str = 'self', decoherent_cc: dict | None = None*)

Specify self-broadening (such as collisional broadening) of a group of states.

Equivalent to calling *add_decoherence_group()* and specifying both state groups to be the same, with the "self" label. For more complicated systems, it may be useful to use `label` to label the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Note that this function applies decoherence terms to every combination of states in the group, not just from each state to its self.

Parameters

- **states** (*list of State*) – List of states to which the self-broadening is applied.

- **gamma** (*ScannableParameter*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. By default, decay will be stored on the graph edge as `"gamma_self"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`
- **decoherent_cc** (*dict, optional*) – Clebsch-Gordon-like coefficients for how gamma scales to different pairs of states within the group. Unspecified pairs are assumed to have coefficients of 0. Default value is None, which applies 0 to all coefficients.

add_single_coupling(*states: Tuple[A_QState, A_QState], rabi_frequency: float | List[float] | ndarray | None = None, detuning: float | List[float] | ndarray | None = None, transition_frequency: float | None = None, phase: float | List[float] | ndarray | None = None, kunit: Sequence[float] = (0, 0, 0), time_dependence: Callable[[float], complex] | None = None, label: str | None = None, e_field: float | List[float] | ndarray | None = None, beam_power: float | None = None, beam_waist: float | None = None, coherent_cc: float | None = None, q: Literal[-1, 0, 1] = 0, kmag_detuning_correction: float | None = None, **extra_kwargs*) → None

Overload of `add_single_coupling()`, which allows for alternate specifications and automatic calculations of some parameter.

This overload fundamentally works identically the `super` method in `Sensor`, with several additions to the functionality that make some assumptions about the underlying system. Because of this, it still preferred to call `add_coupling()` on a `Cell` as well. Please refer to that methods documentation for further detail

Rabi frequency is a mandatory argument in `Sensor` but in `Cell`, there are 3 options for laser power specification:

1. Explicit rabi-frequency definition identical to `Sensor`.
2. Electric field strength, in V/m.
3. Specification of both beam power and beam waist, in W and m respectively.

Any one of these options can be used in place of the standard `rabi_frequency` argument of `add_coupling()`. Note that in all cases, a `rabi_frequency` will be computed and passed to `add_single_coupling()`, and none of the above arguments will be added to the graph. Note that in any of these cases, if the computed dipole moment for the transition is zero, the coupling will be left off the graph.

In all cases, the relative coupling strengths between sublevels of states (if present) is calculated and saved as the `coherent_cc` parameter on the graph. These coefficients are defined to be in units of $1/2\langle J||d||J'\rangle$, as calculated by `getReducedMatrixElementJ()`. The corresponding Rabi frequency that Cell calculates therefore corresponds to $\Omega_{red} = E\langle J||d||J'\rangle/2\hbar$. The Rabi frequency for each transition added to the hamiltonian is then given by

Parameters

- **states** (*sequence*) – Length-2 list-like object (list or tuple) of integers corresponding to the numbered states of the cell. Tuple order indicates which state to has higher energy: namely the second state is always assumed to have higher energy. This order must match the actual energy levels of the atom.
- **rabi_frequency** (*float, optional*) – The rabi frequency, in Mrad/s, of the coupling field. If specified, `e_field`, `beam_power`, and `beam_waist` cannot be specified.
- **detuning** (*float, optional*) – Field detuning, in Mrad/s, of a coupling in the RWA. If specified, RWA is assumed, otherwise RWA not assumed, and transition frequency will be calculated based on atomic properties.
- **transition_frequency** (*float, optional*) – Kept such that method signature matches parent. Value must be `None` as the transition frequency is calculated from atomic properties.
- **phase** (*float, optional*) – Static phase offset in the rotating frame. Cannot be used outside the rotating frame, ie when detuning is not defined. Default is undefined, which is interpreted as 0 for couplings in the rotating frame.

- **kunit** (*sequence, optional*) – A three-element iterable that defines the propagation direction of the field. It should be a normalized vector. This differs from `Sensor`'s `kvec` parameter, since appropriate scale factors are applied automatically in `Cell`. If equal to `(0,0,0)`, solvers will ignore doppler shifts on this field. Defaults to `(0,0,0)`.
- **time_dependence** (*scalar function, optional*) – A scalar function that specifies a time-dependent field. The time dependence function is defined as a function that returns a unitless value as a function of time that is multiplied by the `rabi_frequency` parameter.
- **label** (*str, optional*) – The user-defined name of the coupling. This does not change any calculations, but can be used to track individual couplings, and will be reflected in the output of `axis_labels()` Default None results in using the states tuple as the label.
- **e_field** (*float, optional*) – Electric field strength of the coupling in Volts/meter. If specified, `rabi_frequency`, `beam_power`, and `beam_waist` cannot be specified.
- **beam_power** (*float, optional*) – Beam power in Watts. If specified, `beam_waist` must also be supplied, and `rabi_frequency` and `e_field` cannot be specified. `beam_power` and `beam_waist` cannot be scanned simultaneously.
- **beam_waist** (*float, optional*) – 1/e^2 Beam waist (radius) in units of meters. Only necessary when specifying `beam_power`.
- **q** (*int, optional*) – Coupling polarization in spherical basis. Valid values are -1, 0, 1 for $-\sigma$, linear, $+\sigma$. Default is 0 for linear.
- **kmag_detuning_correction** (*float, optional*) – Detuning to use when calculating the magnitude of the k-vector. By default, `Cell` uses the transition frequency to define the k-vector. For large detuned couplings, this can lead to inaccurate results. Detuning should be given in units of Mrad/s.
- ****extra_kwargs** – Keyword arguments that are passed directly to `Sensor.add_single_coupling()`.

Raises

- ***RydiquleError*** – If `states` is not a list-like of 2 integers.
- ***RydiquleError*** – If an invalid combination of `rabi_frequency`, `e_field`, `beam_power`, and `beam_waist` is provided.
- ***RydiquleError*** – If `transition_frequency` is passed as an argument (it is calculated from atomic properties).
- ***RydiquleError*** – If `beam_power` and `beam_waist` are both sequences.
- ***CouplingNotAllowedError*** – If the coupling is not dipole-allowed.
- ***RydiquleError*** – If `kvec` is supplied instead of `kunit`.

Notes

Note

Note that while this function can be used directly just as in *Sensor*, it will often be called implicitly via `add_coupling()` which `Cell` inherits. While they are equivalent, the second of these options is often the more clear approach, and it automatically sets the `probe_tuple` attribute.

Note

Specifying the beam power by beam parameters or electric field still computes the `rabi_frequency` and adds that quantity to the `Cell` to maintain consistency across `rydiqule`'s other calculations. In other words, `beam_power`, `beam_waist`, and `e_field` will never appear as quantities on the graph of a `Cell`.

Examples

In the simplest case, physical properties are calculated automatically in a `Cell` All the familiar quantities are present, as well as many more. Note that while not strictly necessary, it is often convenient to alias states with shorthand variables to avoid very cumbersome state specification.

```
>>> [g, e] = rq.D2_states("Rb87")
>>> cell = rq.Cell("Rb87", [g, e])
>>> cell.add_single_coupling((g,e), detuning=1.0, rabi_frequency=2.0, label=
↪"probe")
>>> print(cell)
<class 'rydiqule.cell.Cell'> object with 2 states and 1 coherent couplings.
States: [(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
Coherent Couplings:
    ((5, 0, 0.5),(5, 1, 1.5)): {rabi_frequency: 2.0, detuning: 1.0, phase: 0,␣
↪kvec: (0, 0, 0), label: probe, coherent_cc: 1, dipole_moment: 2.44, q: 0}
Decoherent Couplings:
    ((5, 0, 0.5),(5, 0, 0.5)): {gamma_transit: 0.40697}
    ((5, 1, 1.5),(5, 0, 0.5)): {gamma_transition: 38.11316, gamma_transit: 0.
↪40697}
Energy Shifts:
    None
```

Since *add_couplings()*, *add_coupling()*, and *add_coupling_group()* only iterate over calls of this function, they do not need to be overloaded.

```
>>> [g, e] = rq.D2_states("Rb87")
>>> cell = rq.Cell("Rb87", [g, e])
>>> probe = dict(states=(g,e), detuning=1.0, rabi_frequency=2.0, label="probe")
>>> cell.add_couplings(probe)
>>> print(cell)
<class 'rydiqule.cell.Cell'> object with 2 states and 1 coherent couplings.
States: [(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
Coherent Couplings:
    ((5, 0, 0.5),(5, 1, 1.5)): {rabi_frequency: 2.0, detuning: 1.0, phase: 0,␣
↪kvec: (0, 0, 0), label: probe, coherent_cc: 1, dipole_moment: 2.44, q: 0}
Decoherent Couplings:
    ((5, 0, 0.5),(5, 0, 0.5)): {gamma_transit: 0.40696}
    ((5, 1, 1.5),(5, 0, 0.5)): {gamma_transition: 38.113, gamma_transit: 0.
↪40696}
Energy Shifts:
    None
```

`e_field` can be specified instead of `rabi_frequency`, but a `rabi_frequency` will still be added to the system based on the `e_field` and computed dipole moment rather than `e_field` directly.

```
>>> [g, e] = rq.D2_states("Rb87")
>>> cell = rq.Cell("Rb87", [g, e])
>>> cell.add_coupling((g,e), detuning=1.0, e_field=6.0, label="probe")
>>> print(cell)
<class 'rydiqule.cell.Cell'> object with 2 states and 1 coherent couplings.
States: [(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
Coherent Couplings:
    ((5, 0, 0.5),(5, 1, 1.5)): {rabi_frequency: 1.177, detuning: 1.0, phase: 0,
↪ kvec: (0, 0, 0), label: probe, coherent_cc: 1, dipole_moment: 2.44, q: 0}
Decoherent Couplings:
    ((5, 0, 0.5),(5, 0, 0.5)): {gamma_transit: 0.40696}
    ((5, 1, 1.5),(5, 0, 0.5)): {gamma_transition: 38.11, gamma_transit: 0.
↪40696}
Energy Shifts:
    None
```

As can `beam_power` and `beam_waist`, with similar behavior regarding how information is stored.

```
>>> cell = rq.Cell("Rb85", rq.D2_states("Rb85"), cell_length = .0001)
>>> cell.add_coupling((g,e), detuning=1.0, beam_power=1.0, beam_waist=1.0,␣
↪label="probe")
>>> print(cell)
<class 'rydiqule.cell.Cell'> object with 2 states and 1 coherent couplings.
States: [(n=5, l=0, j=0.5), (n=5, l=1, j=1.5)]
Coherent Couplings:
    ((5, 0, 0.5),(5, 1, 1.5)): {rabi_frequency: 4.3, detuning: 1.0, phase: 0,␣
↪kvec: (0, 0, 0), label: probe, coherent_cc: 1, dipole_moment: 2.44, q: 0}
Decoherent Couplings:
    ((5, 0, 0.5),(5, 0, 0.5)): {gamma_transit: 0.4117}
    ((5, 1, 1.5),(5, 0, 0.5)): {gamma_transition: 38.113, gamma_transit: 0.
↪4117}
Energy Shifts:
    None
```

add_single_decoherence(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, decoherent_cc: float = 1.0, label: str | None = None*)

Add decoherent coupling to the graph between two states.

If `gamma` is list-like, the array generated by `decoherence_matrix()` will contain decoherence matrices for every combination of decoherence values provided. This functionality mirrors hamiltonian generation when parameters of `add_coupling()` are list-like. Note that if `gamma` is 0 or an array of zeros, the associated edge key will be left off the graph.

Parameters

- **states** (*tuple of State*) – Length-2 tuple of integers corresponding to the two states. The first value is the number of state out of which population decays, and the second is the number of the state into which population decays.
- **gamma** (*float or sequence*) – The decay rate, in Mrad/s.
- **decoherent_cc** (*float*) – The value by which `gamma` is multiplied before it is added to the graph. Typically only used by `add_decoherence_group()`, but made transparent for scripting purposes. Defaults to 1.0
- **label** (*str or None, optional*) – Optional label for the decay. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`.

Notes

Note

Adding a decoherence with a particular label (including `None`) will override an existing decoherent transition with that label.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), 0.1, label="misc")
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.  0. ]
 [0.1 0.  0. ]]
```

To add multiple decoherence effects to the same term, provide a different label for each.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), 0.1, label='foo')
>>> s.add_single_decoherence((2,0), 0.15, label='bar')
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.   0.   0.  ]
 [0.25 0.   0.  ]]
```

Decoherence values can also be scanned. Here decoherence from states 2->0 is scanned between 0 and 0.5 for 11 values. We can also see how the Hamiltonian shape accounts for this to allow for clean broadcasting, indicating that the hamiltonian is identical across all decoherence values.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

add_single_energy_shift(*state: int | str | Tuple[float, ...], shift: float | List[float] | ndarray, label=None*)

Add an energy shift to a state.

First performs validation that the provided `state` is actually a node in the graph, then adds the shift specified by `shift` to a self-loop edge keyed with `"e_shift"`. This value will be added to the corresponding diagonal term when the hamiltonian is generated.

Parameters

- **state** (*int, str, or tuple*) – The label corresponding to the atomic state to which the shift will be added.
- **shift** (*float or list-like of float*) – The magnitude of the energy shift, in Mrad/s

Raises

RydiquleError – If the supplied `state` is not in the system.

add_transit_broadening(*gamma_transit: float | List[float] | ndarray, repop: Dict[A_QState, float] | List[A_QState] | None = None, label: str = 'transit'*)

Overload of the `add_transit_broadening()` method of `Sensor` which automatically populates the `repop` dictionary with a equal decay to all sublevels of the ground `(n, l, j)` state.

Parameters

- **gamma_transit** (*ScannableParameter*) – Transit broadening of the system. Passed transparently to super function.
- **repop** (*dict, optional*) – Dictionary of states for transit to repopulate in to. The keys represent tshe state labels. The values represent the fractional amount that goes to that state. If the sum of value does not equal 1, population will not be conserved. If `None`, raises population decay due to transit broadening will be evenly divided amongst all states matching the `(n, l, j)` quantum numbers of the lowest-energy state in the system. Defaults to `None`.
- **label** (*str, optional*) – Label to be passed to `add_decoherence()`. Defaults to "transit"

Examples

```
>>> atom = "Rb85"
>>> g = rq.ground_state(atom, splitting="fs")
>>> e = rq.D1_excited(atom, splitting="fs")
>>> Rb_Cell = rq.Cell(atom, [g,e])
>>> Rb_Cell.add_transit_broadening(0.1)
>>> for e in Rb_Cell.couplings.edges.data("gamma_transit"):
...     print(e)
((n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=-0.5), 0.05)
((n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), 0.05)
((n=5, l=0, j=0.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=-0.5), 0.05)
((n=5, l=0, j=0.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=0.5), 0.05)
((n=5, l=1, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=-0.5), 0.05)
((n=5, l=1, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5), 0.05)
((n=5, l=1, j=0.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=-0.5), 0.05)
((n=5, l=1, j=0.5, m_j=0.5), (n=5, l=0, j=0.5, m_j=0.5), 0.05)
```

atom_mass: float = None

Mass of an atom in the vapor cell, in kilograms.

axis_labels() → List[str]

Get a list of axis labels for stacked hamiltonians.

The axes of a hamiltonian stack are defined as the axes preceding the usual hamiltonian, which are always the last 2. These axes only exist if one of the parameters used to define a Hamiltonian are lists.

Be default, labels which have been zipped using *zip_parameters()* will be combined into a single label, as this is how *get_hamiltonian()* treats these axes.

The ordering of axis labels is as follows:

- Zipped parameter (shared axes) appear before single parameters.
- Zipped parameters are ordered alphabetically by label.
- Single axes are sorted first by lower state, then by upper state, then alphabetically by parameter.

Returns

Strings corresponding to the label of each axis on a stack of multiple hamiltonians.

Return type

list of str

Examples

There are no preceding axes if there are no list-like parameters.

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(3, 3)
>>> print(s.axis_labels())
[]
```

Adding list-like parameters expands the hamiltonian

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det, "label":"blue"}
```

(continues on next page)

(continued from previous page)

```
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
>>> print(s.axis_labels())
['blue_detuning', '(1,2)_detuning']
```

The ordering of labels doesn't change if string state names are used. For single couplings, the ordering of axes is determined purely by the ordering of the states, regardless of coupling labels or string names of states.

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":('g','e1'), "rabi_frequency":1, "detuning":det, "label":
↪"blue"}
>>> red = {"states":('e1','e2'), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
>>> print(s.axis_labels())
['blue_detuning', '(e1,e2)_detuning']
```

Zipping parameters combines labels onto a single axis, since their Hamiltonians now lie on a single axis of the stack. The name of that axis will be the label provided to `zipped_parameters()`. Note that this will default to 'zip_<int>'. Here the axis of length 7 (axis 1) corresponds to the rabi frequencies and the axis of shape 11 (axis 0) corresponds to the zipped detunings

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=np.
↪linspace(-3, 3, 7))
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_frequency=1)
>>> s.zip_parameters({(0,1):"detuning", (1,2):"detuning"}, zip_label="detunings
↪")
>>> print(s.get_hamiltonian().shape)
(11, 7, 3, 3)
>>> print(s.axis_labels())
['detunings', '(0,1)_rabi_frequency']
```

property basis_size: int

Property to return the number of nodes on the Sensor graph.

Returns

The number of nodes on the graph, corresponding to the basis size for the system.

Return type

int

beam_area: float | None = None

Cross-sectional area of the probing beam, in square meters.

cell_length: float | None = None

Optical path length of the medium, in meters.

coupling_subgraph(*coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]]*) → Graph

Returns a subgraph view of the couplings graph corresponding to `coupling`.

Parameters

coupling (*StateSpecs*) – Coupling specification

Returns
View of the corresponding subgraph

Return type
networkx.Graph

couplings_with(**keys: str, method: Literal['all', 'any', 'not any'] = 'all'*) → Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Dict]

Returns a version of self.couplings with only the keys specified.

Can be specified with a several criteria, including all, none, or any of the keys specified.

Parameters

- **str)** (*keys (tuple of*) – parameter names for a state. See `add_coupling()` for which names are valid for a Sensor object.
- **method** (*{'all','any', 'not any'}*) – Method to see if a given field matches the keys given. Choosing "all" will return couplings which have keys matching all of the values provided in the keys argument, while choosing "any", will return all couplings with keys matching at least one of the values specified by keys. For example, `sensor.couplings_with("rabi_frequency")` returns a dictionary of all couplings for which a rabi_frequency was specified. `sensor.couplings_with("rabi_frequency", "detuning", method="all")` returns all couplings for which both rabi_frequency and detuning are specified. 'sensor.couplings_with("rabi_frequency", "detuning", method="any")` returns all couplings for which either rabi_frequency or detuning are specified. Defaults to "all".

Returns
A copy of the `sensor.couplings` dictionary with only couplings containing the specified parameter keys.

Return type
dict

Examples

Can be used, for example, to return couplings in the rotating wave approximation.

```
>>> s = rq.Sensor(3)
>>> sinusoid = lambda t: 0 if t<1 else sin(100*t)
>>> f2 = {"states": (0,1), "detuning": 1, "rabi_frequency":2}
>>> f1 = {"states": (1,2), "transition_frequency":100, "rabi_frequency":1,
↪"time_dependence": sinusoid}
>>> s.add_couplings(f1, f2)
>>> gamma = np.array([[.2,0,0],
...                   [.1,0,0],
...                   [0.05,0,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.couplings_with("detuning"))
{(0, 1): {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0, 0),
↪'coherent_cc': 1.0, 'label': '(0,1)'}}
```

decoherence_matrix() → ndarray

Build a decoherence matrix out of the decoherence terms of the graph.

For each edge, sums all parameters with a key that begins with "gamma", and places it on the appropriate location in an adjacency matrix for the `couplings` graph.

Returns
The decoherence matrix stack of the system.

Return type
numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_decoherence((1,0), 0.2, label="foo")
>>> s.add_decoherence((1,0), 0.1, label="bar")
>>> s.add_decoherence((2,0), 0.05)
>>> s.add_decoherence((2,1), 0.05)
>>> print(s.couplings.edges(data=True))
[(1, 0, {'gamma_foo': 0.2, 'label': '(1,0)', 'gamma_bar': 0.1}), (2, 0, {'gamma
↪': 0.05, 'label': '(2,0)'}), (2, 1, {'gamma': 0.05, 'label': '(2,1)'})]
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.3  0.   0.  ]
 [0.05 0.05 0.  ]]
```

Decoherences can be stacked just like any parameters of the Hamiltonian:

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5, 11)
>>> s.add_decoherence((1,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
```

Defining decoherences between states labelled with string values works just like coherent couplings:

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> s.add_decoherence(('e1', 'g'), 0.1)
>>> s.add_decoherence(('e2', 'g'),0.1)
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.1 0.  0. ]
 [0.1 0.  0. ]]
```

dm_basis() → ndarray

Generate basis labels of density matrix components.

The basis corresponds to the elements in the solution. This is not the complex basis of the sensor class, but rather the real basis of a solution after calling one of `rydiqule`'s solvers. This means that the ground state population has been removed and it has been transformed to the real basis.

Returns

Array of string labels corresponding to the solving basis. Is a 1-D array of length `n**2-1`.

Return type

numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> print(s.dm_basis())
['10_real' '20_real' '10_imag' '11_real' '21_real' '20_imag' '21_imag'
 '22_real']
```

property eta: float

Get the eta value for the system.

The value is computed with the following formula Eq. 7 of Meyer et. al. PRA 104, 043103 (2021)

$$\eta = \sqrt{\frac{\omega \mu^2}{2 c \epsilon_0 \hbar A}}$$

Where ω is the probing frequency, μ is the dipole moment, A is the beam area, c is the speed of light, ϵ_0 is the dielectric constant, and $\hbar$ is the reduced Plank constant.

This value is only computed if there is not a `_eta` attribute in the system. If this attribute does exist, this function acts as an accessor for that attribute.

Returns
The value eta for the system.

Return type
float

get_couplings() → Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Dict]

Returns the couplings of the system as a dictionary

Deprecating in favor of calling the couplings.edges attribute directly.

Returns
A dictionary of key-value pairs with the keys corresponding to levels of transition, and the values being dictionaries of coupling attributes.

Return type
dict

get_doppler_shifts() → ndarray

Returns the Hamiltonian with only detunings set to the most probable doppler shift values for each spatial dimension.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns
Array of shape (used_spatial_dim,n,n), Hamiltonians with only the doppler shifts present along each non-zero spatial dimension specified by the fields' "kvec" parameter.

Return type
numpy.ndarray

get_hamiltonian() → ndarray

Creates the Hamiltonians from the couplings defined by the fields.

They will only be the steady state hamiltonians, i.e. will only contain terms which do not vary with time. Implicitly creates hamiltonians in "stacks" by creating a grid of all supported coupling parameters which are lists. This grid of parameters will not contain rabi-frequency parameters which vary with time and are defined as list-like. Rather, the associated axis will be of length 1, with the scanning over this value handled by the `get_time_couplings()` function.

For m list-like parameters x1,x2,…,xm with shapes N1,N2,…,Nm, and basis size n, the output will be shape `(N1,N2,...,Nm, n, n)`. The dimensions N1,N2,…Nm are labeled by the output of `axis_labels()`.

If any parameters have been zipped with the `_zip_parameters()` method, those parameters will share an axis in the final hamiltonian stack. In this case, if axis N1 and N2 above are the same shape and zipped, the final Hamiltonian will be of shape `(N1,...,Nm, n, n)`.

In the case where the basis of the `Sensor` was explicitly defined with a list of states, the ordering of rows and columns in the hamiltonian corresponds to the ordering of states passed in the basis.

See rydiqule's conventions for matrix stacking for more details.

Returns
The complex hamiltonian stack for the sensor.

Return type
np.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
```

Time dependent couplings are handled separately. The axis that contains array-like parameters with time dependence is length 1 in the steady-state Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
```

Zipping parameters means they share an axis in the Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=2)
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_frequency=1)
>>> s.zip_parameters({(0,1):"detuning", (1,2):"detuning"})
>>> H = s.get_hamiltonian()
>>> print(H.shape)
(11, 3, 3)
```

If the basis is provided as a list of string labels, the ordering of Hamiltonian rows and columns will correspond to the order of states provided.

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> s.add_coupling(('g', 'e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(('e1', 'e2'), detuning=1.5, rabi_frequency=1)
>>> print(s.get_hamiltonian())
[[ 0. +0.j  0.5+0.j  0. +0.j]
 [ 0.5-0.j -1. +0.j  0.5+0.j]
 [ 0. +0.j  0.5-0.j -2.5+0.j]]
```

get_hamiltonian_diagonal(*values: dict*, *no_stack: bool = False*) → ndarray

Apply addition and subtraction logic corresponding to the direction of the couplings.

For a given state `n`, the path from ground will be traced to `n`. For each edge along this path, values will be added where the path direction and coupling direction match, and subtracting values where they do not. The sum of all such values along the path is the `n` th term in the output array.

Designed for internal functions which help generate hamiltonians. Most commonly used to calculate total detunings for ranges of couplings under the RWA

Parameters

- **values** (*dict*) – Key-value pairs where the keys correspond to transitions (agnostic to ordering of states) and values corresponding to the values to which the logic will be applied.
- **no_stack** (*bool, optional*) – Whether to ignore variable parameters in the system and use only basic math operations rather than reshape the output. Typically only `True` for calculating

doppler shifts.

Returns
The diagonal of the hamiltonian of the system of shape `(*l,n)`, where `l` is the shape of the hamiltonian stack for the sensor.

Return type
numpy.ndarray

get_parameter_mesh() → List[ndarray]

Returns the parameter mesh of the sensor.

The parameter mesh is the flattened grid of variable parameters in all the couplings of a sensor. Wraps `numpy.meshgrid` with the `indexing` argument always `"ij"` for matrix indexing.

Returns
list of mesh grids for every variable parameter

Return type
list of numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> rabi1 = np.linspace(-1,1,11)
>>> rabi2 = np.linspace(-2,2,21)
>>> s.add_coupling(states=(0,1), rabi_frequency=rabi1, detuning=1)
>>> s.add_coupling(states=(1,2), rabi_frequency=rabi2, detuning=1)
>>> for p in s.get_parameter_mesh():
...     print(p.shape)
(11, 1)
(1, 21)
```

get_rotating_frames() → dict

Determines the rotating frames for the disconnected subgraphs.

Each returned path gives the states traversed, and the sign gives the direction of the coupling. If the sign is negative, the coupling is going to a lower energy state. Choice of frame depends on graph distance to lowest indexed node on subgraph, ties broken by lowest indexed path traversed first.

Returns
Dictionary keyed by disconnected subgraphs, values are path dictionaries for each node of the subgraph. Each path shows the node indexes traversed, where a negative sign denotes a transition to a lower energy state.

Return type
dict

get_time_dependence() → List[Callable[[float], complex]]

Function which returns a list of the `time_dependence` functions.

The list is returned with in the order that matches with the time hamiltonians from `get_time_couplings()` such that the ith element of of the return of this functions corresponds with the ith Hamiltonian terms returned by that function.

Returns
List of scalar functions, representing all couplings specified with a `time_dependence`.

Return type
list

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> print(s.get_time_dependence())
[<function <lambda> at ...>, <function <lambda> at ...>]
```

get_time_hamiltonian(*t: float*) → ndarray

Get the system hamiltonian at a specific time, t.

This sums the steady-state hamiltonians with the time-dependent parts, evaluated at a specific time, t. If there is no time dependence in the system, function is equivalent to *get_hamiltonian()*.

Parameters

t (*float*) – Time to evaluate the time-dependence function at when building the hamiltonians

Returns

System hamiltonian, evaluated at time t.

Return type

numpy.ndarray

get_time_hamiltonian_components() → Tuple[List[ndarray], List[ndarray]]

Get time-dependent components of the hamiltonian.

Returns the list of matrices of all couplings in the system defined with a `time_dependence` key. The output will be two lists of matrices representing terms of the hamiltonian which are dependent on each time-dependent coupling. The lists will be of length M and shape `(*l_time, n, n)`, where M is the number of time-dependent couplings, `l_time` is time-dependent stack shape (possibly all ones), and `n` is the basis size. Each matrix will have terms equal to the rabi frequency (or half the rabi frequency under RWA) in positions that correspond to the associated transition. For example, in the case where there is a `time_dependence` function defined for the `(2,3)` transition with a rabi frequency of 1, the associated time coupling matrix will be all zeros, with a 1 in the `(2,3)` and `(3,2)` positions.

Typically, this function is called internally and multiplied by the output of the *get_time_dependence()* function.

Returns

- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the real-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.
- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the imaginary-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
```

(continues on next page)

(continued from previous page)

```
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> time_hams, time_hams_i = s.get_time_hamiltonian_components()
>>> for H in time_hams:
...     print(H)
[[0.+0.j 1.+0.j 0.+0.j]
 [1.-0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j]]
[[0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 2.+0.j]
 [0.+0.j 2.-0.j 0.+0.j]]
```

To handle stacking across the steady-state and time hamiltonians, the dimensions are matched in a way that broadcasting works in a numpy-friendly way

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> time_hams, time_hams_i = s.get_time_hamiltonian_components()
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
>>> print(time_hams[0].shape)
(1, 11, 3, 3)
>>> print(time_hams_i[0].shape)
(1, 11, 3, 3)
```

get_transition_frequencies() → ndarray

Gets an array of the diagonal elements of the Hamiltonian from the field detunings.

Wraps the *get_hamiltonian_diagonal()* function using both transition frequencies and detunings. Primarily for internal use.

Returns

N-D array of the hamiltonian diagonal. For an n-level system with stack shape `*l`, will be shape `(*l, n)`

Return type

numpy.ndarray

get_value_dictionary(*key: str*) → dict

Get subset of dictionary coupling parameters.

Return a dictionary of key value pairs where the keys are couplings added to the system and the values are the value of the parameter specified by key. Produces an output that can be passed directly to *get_hamiltonian_diagonal()*. Only couplings whose parameter dictionaries contain "key" will be in the returned dictionary.

Parameters

key (*str*) – String value of the parameter name to build the dictionary. For example, `get_value_dictionary("detuning")` will return a dictionary with keys corresponding to transitions and values corresponding to detuning for each transition which has a detuning.

Returns

Coupling dictionary with couplings as keys and corresponding values set by input key.

Return type
dict

Examples

```
>>> s = rq.Sensor(4)
>>> f1 = {"states": (0,1), "detuning": 2, "rabi_frequency": 1}
>>> f2 = {"states": (1,2), "detuning": 3, "rabi_frequency": 2}
>>> step = lambda t: 1 if t>1 else 0
>>> f3 = {"states": (2,3), "rabi_frequency": 3, "transition_frequency": 3,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2, f3)
>>> print(s.get_value_dictionary("detuning"))
{(0, 1): 2, (1, 2): 3}
```

group_variable_parameters(*apply_mesh: bool = False*) → List[List[Tuple[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], str, ndarray, str | None]]]

int_states_map(*invert: bool = False*) → Dict[int | str | Tuple[float, ...], int] | Dict[int, int | str | Tuple[float, ...]]

Get a dictionary mapping between state labels and their corresponding integer ordering. Can be returned with `key:value` pairs defined either by `label:int` or `int:label`, controlled via optional `invert` argument.

Parameters
invert (*bool, optional*) – Whether to switch the role of keys and values. Labels are keys if `False`, and values if `True`, by default False

Returns
Dictionary mapping between state labels and integer ordering

Return type
dict

property kappa: float

Property to calculate the kappa value of the system.

The value is computed with the following formula Eq. 5 of Meyer et. al. PRA 104, 043103 (2021)

$$\kappa = \frac{\omega n \mu^2}{2c\epsilon_0\hbar}$$

Where ω is the probing frequency, μ is the dipole moment, n is atomic cloud density, c is the speed of light, ϵ_0 is the dielectric constant, and $\hbar$ is the reduced Plank constant.

This value is only computed if there is not a `_kappa` attribute in the system. If this attribute does exist, this function acts as an accessor for that attribute.

Returns
The value kappa for the system.

Return type
float

level_ordering() → List[*A_QState*]

Return a list of the states in the `Cell` in ascending energy order.

All energies are calculated with respect to the ground state energy, which is defined as 0. Ground state is determined by the rydiqule's calculation of ground energy, which uses `arc` to get the energy of the $nP^{\frac{1}{2}}$ state, where `n` is 1 for Hydrogen, 2 for Lithium, etc.

Returns
The Cell states in order of descending energy relative to the ground state $nS^{\frac{1}{2}}$.

Return type
list of *A_QState*

Examples

For the following example, states are in the list passed to the constructor in ascending energy order, so the ordering the basis is identical to the `level_ordering`. Computed `Cell` attributes the Hamiltonian will, for clarity, always appear in the ordering of levels in the list passed to the constructor.

```
>>> from rydiqule import A_QState
>>> atom = "Rb85"
>>> [g, e] = rq.D2_states(atom)  #uses the D2 line of Rb85
>>> state1 = A_QState(50, 2, 2.5)
>>> state2 = A_QState(51, 2, 2.5)
>>> my_cell = rq.Cell(atom, [g, e, state1, state2])
>>> print(my_cell.states)
[(n=5, l=0, j=0.5), (n=5, l=1, j=1.5), (n=50, l=2, j=2.5), (n=51, l=2, j=2.5)]
>>> # levels in order
>>> for i, state in enumerate(my_cell.level_ordering()):
...     print(f"{i}: {state}, E={my_cell.couplings.nodes[state]['energy']*2*np.
↪pi*1e-6} Mrad/s")
0: (5, 0, 0.5), E=0.0 Mrad/s
1: (5, 1, 1.5), E=15168.8 Mrad/s
2: (50, 2, 2.5), E=39819.3 Mrad/s
3: (51, 2, 2.5), E=39821.5 Mrad/s
```

If we scramble the states in the constructor, the output of this function remains the same even though the order of basis states changes to match the list ordering in the constructor.

```
>>> from rydiqule import A_QState
>>> atom = "Rb85"
>>> [g, e] = rq.D2_states(atom)  #uses the D2 line of Rb85
>>> state1 = A_QState(50, 2, 2.5)
>>> state2 = A_QState(51, 2, 2.5)
>>> my_cell = rq.Cell(atom, [state2, e, g, state1])
>>> print(my_cell.states)
[(n=51, l=2, j=2.5), (n=5, l=1, j=1.5), (n=5, l=0, j=0.5), (n=50, l=2, j=2.5)]
>>> for i, state in enumerate(my_cell.level_ordering()):
...     print(f"{i}: {state}, E={my_cell.couplings.nodes[state]['energy']*2*np.
↪pi*1e-6} Mrad/s")
0: (5, 0, 0.5), E=0.0 Mrad/s
1: (5, 1, 1.5), E=15168.8 Mrad/s
2: (50, 2, 2.5), E=39819.3 Mrad/s
3: (51, 2, 2.5), E=39821.5 Mrad/s
```

property probe_freq: float

Get the probe transition frequency, in rad/s.

Note that for *Cell*, probing transition frequency is calculated using only the `(n,l,j)` states of the upper and lower manifolds of the `probe_tuple` attribute. For more precise calculations accounting for atomic splitting etc, `probe_freq` must be set manually; `rydiqule` does not support doing these calculations automatically.

Returns
Probe transition frequency, in rad/s, between probing nlj states.

Return type
float

property probe_tuple: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]] | None

Coupling edge that corresponds to the probing field. Defaults to `None` and gets set to the first coupling added to the system with *add_coupling()*. Can be modified directly.

set_experiment_values(*probe_freq: float*, *kappa: float*, *eta: float | None = None*, *cell_length: float | None = None*, *beam_area: float | None = None*)

`Sensor` specific method. Do not use with `Cell`.

This function does not do anything as Cell automatically handles this functionality internally.

Warns

RydiqueWarning (*Warns if function is used.*)

set_gamma_matrix(*gamma_matrix: ndarray*)

Set the decoherence matrix for the system.

Works by first removing all existing decoherent data from graph edges, then individually adding all nonzero terms of a provided gamma matrix to the corresponding graph edges. Can be used to set all decoherence attributes to edges simultaneously, but `add_decoherence()` is preferred.

Unlike `add_decoherence()`, does not support scanning multiple decoherence values, rather should be used to set the decoherences of the system to individual static values.

Parameters

gamma_matrix (*numpy.ndarray*) – Array of shape `(basis_size, basis_size)`. Element `(i,j)` describes the decoherence rate, in Mrad/s, from state `i` to state `j`.

Raises

- **RydiquleError** – If `gamma_matrix` is not a numpy array.
- **ValueError** – If `gamma_matrix` is not a square matrix of the appropriate size
- **ValueError** – If the shape of `gamma_matrix` is not compatible with `self.basis_size`.

Examples

```
>>> s = rq.Sensor(2)
>>> f1 = {"states": (0,1), "detuning":1, "rabi_frequency": 1}
>>> s.add_couplings(f1)
>>> gamma = np.array([[.1,0],[.1,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.decoherence_matrix())
[[0.1 0. ]
 [0.1 0. ]]
```

spatial_dim() → int

Returns the number of spatial dimensions doppler averaging will occur over.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns

Number of dimensions, between 0 and 3, where 0 means no doppler averaging k-vectors have been specified or are too small to be calculates.

Return type

int

Examples

No spatial dimensions specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1)
>>> print(s.spatial_dim())
0
```

One spatial dimension specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(0,0,4))
>>> print(s.spatial_dim())
1
```

Multiple spatial dimensions can exist in a single coupling or across multiple couplings

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(3,0,3))
>>> print(s.spatial_dim())
2
```

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(3,0,3))
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, kvec=(0,4,0))
>>> print(s.spatial_dim())
3
```

property states: List[int | str | Tuple[float, ...]]

Property which gets a list of labels for the sensor in the order defined in `__init__()`. This is also the order corresponding the rows and columns in the system Hamiltonian and decoherence matrix.

Returns
List of states of the system defined the constructor, in the order corresponding to rows and columns of the Hamiltonian.

Return type
list

states_with_spec(*statespec: A_QState*) → List[*A_QState*]

Return a list of all states in the sensor matching the `state_spec` pattern.

Matching is determined by same rules as `states_with_spec()`, with no additional logic to account for the different typing of the states. This means that there is expansion of any quantum numbers specified by the "all" keyword, and only states that are already nodes of the `Cell` graph will be included.

Parameters
statespec (`A_QState`) – State specification against which to perform matching,

Returns
List of all states in `Cell` instance which match the provided specification.

Return type
List[*A_QState*]

Examples

```
>>> atom = "Rb85"
>>> g = rq.ground_state(atom, splitting="fs")
>>> e = rq.D1_excited(atom, splitting="fs")
>>> Rb_Cell = rq.Cell(atom, [g,e])
>>> print(Rb_Cell.states_with_spec(A_QState(5, 0, 0.5)))
[]
>>> print(Rb_Cell.states_with_spec(A_QState(5, 0, 0.5, m_j="all")))
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5)]
>>> print(Rb_Cell.states_with_spec(A_QState(5, 0, 0.5, m_j=[-0.5, 0.5])))
[(n=5, l=0, j=0.5, m_j=-0.5), (n=5, l=0, j=0.5, m_j=0.5)]
```

temp: float | None = None

Temperature of the vapor cell, in Kelvin.

unzip_parameters(*zip_label: str*, *verbose: bool | None = True*)

Remove a set of zipped parameters from the internal zip_labels list.

If an element of the internal `_zip_labels` array matches the label provided, removes it from `_zip_labels`. If no such element is present in `_zip_labels`, does nothing, and prints a message (disabled with `verbose=False`)

Parameters

- **zip_label** (*str*) – The string label corresponding the key to be deleted in the `_zip_labels` attribute.
- **verbose** (*bool*) – Whether to print a message if the unzip fails due to the specified `zip_label` not being a zip in the sensor. If `True` prints a message to std out if `zip_label` is not an element of the internal `self._zip_labels`. Otherwise, fails silently. Can be used if unzipping as part of an automated script.

Notes

Note

This function should always be used rather than modifying the `_zip_labels` attribute directly.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label="probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"}, zip_label="demo1")
>>> print(s._zip_labels) #NOT modifying directly
['demo1']
>>> print(s.couplings.edges(data="demo1"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> s.unzip_parameters("demo1")
>>> print(s._zip_labels) #NOT modifying directly
[]
>>> print(s.couplings.edges(data="demo1"))
[(0, 1, None), (1, 2, None)]
```

If the labels provided are not a match, a message is printed and nothing is altered. In the case where simulations are scripted and the printed message is annoying, the print behavior can be modified with `verbose=False`, potentially useful for scripting cases where the desired behavior is to silently continue over non-existent zip labels.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label="probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"})
>>> print(s._zip_labels) #NOT modifying directly
['zip_0']
>>> print(s.couplings.edges(data="zip_0"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> s.unzip_parameters("zipp0")
```

(continues on next page)

(continued from previous page)

```
No label matching zipp0, no action taken
>>> print(s._zip_labels) #NOT modifying directly
['zip_0']
>>> print(s.couplings.edges(data="zip_0"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
```

property vP: float

Most probable speed of the 3D Maxwell-Boltzmann distribution.

This is defined as $\sqrt{2kT/m}$ and is given in units of m/s.

This must be defined manually when performing Doppler-broadened solves. Accessing it before definition will raise an error.

v_th: float | None = None

Thermal velocity of the atoms in vapor cell, in meters per second.

variable_parameter_sort(*par: tuple*) → tuple

Assistance function which determines the sorting order of elements parameters in sensor.

Called in `variable_parameters()` to ensure a consistent sort order. Provided as the `key` parameter in python's `sorted()` function before parameters are returned.

Sorts first by `zip_label`, then by `states`, then by `parameter`. Ensures all parameters zipped with one another are grouped together in a list. Zipped parameters will always come first. From there, parameters are sorted alphabetically by zip_label (including case), then by state pair (as determined by ordering in the sensor, NOT alphabetically), then alphabetically by parameter.

Parameters

par (*tuple*) – 4-element list of information on each parameter. Consists of `(states, parameter, value, zip_label)`

Returns

3-element tuple that defines a particular parameter's position in the final sorting order.

Return type

tuple

variable_parameters(*apply_mesh: bool = False*) → List[Tuple[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], str, ndarray, str | None]]

Property to retrieve the values of parameters that were stored on the graph as arrays.

Values are returned as a list of tuples in the standard order of pythons default sorting, applied first to the tuple indicating states and then to the key of the parameter itself. This means that couplings are sorted first by lower state, then by upper state, then alphabetically by the name of the parameter.To determine order, all state labels treated as their integer position in the basis as determined by ordering in the constructor `__init__()`.

Returns

A list of tuples corresponding to the parameters of the systems that are variable (i.e. stored as an array). They are ordered according to states, then according to variable name. Tuple entries of the list take the form `(states, param_name, value)`

Return type

list of tuples

Examples

```
>>> s = rq.Sensor(3)
>>> vals = np.linspace(-1,2,3)
>>> s.add_coupling(states=(1,2), rabi_frequency=vals, detuning=1)
>>> s.add_coupling(states=(0,1), rabi_frequency=vals, detuning=vals)
```

(continues on next page)

(continued from previous page)

```
>>> print(s.variable_parameters())
[((0, 1), 'detuning', array([-1. ,  0.5,  2. ]), None),
 ((0, 1), 'rabi_frequency', array([-1. ,  0.5,  2. ]), None),
 ((1, 2), 'rabi_frequency', array([-1. ,  0.5,  2. ]), None)]
```

The order is important; in the unzipped case, it will sort as though all state labels were cast to strings, meaning integers will always be treated as first.

```
>>> s = rq.Sensor([0, 'e1', 'e2'])
>>> det1 = np.linspace(-1, 1, 3)
>>> det2 = np.linspace(-1, 1, 5)
>>> blue = {"states":(0,'e1'), "rabi_frequency":1, "detuning":det1}
>>> red = {"states":('e1','e2'), "rabi_frequency":3, "detuning":det2}
>>> s.add_couplings(blue, red)
>>> print(s.variable_parameters())
[((0, 'e1'), 'detuning', array([-1.,  0.,  1.]), None),
 (('e1', 'e2'), 'detuning', array([-1. , -0.5,  0. ,  0.5,  1. ]), None)]
>>> print(f"Axis Labels: {s.axis_labels()}")
Axis Labels: ['(0,e1)_detuning', '(e1,e2)_detuning']
```

zip_parameters(*parameters: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]] | str, str], zip_label: str | None = None*)

Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Zipped parameters will share an axis when quantities relevant to the equations of motion, such as the `gamma_matrix` and `hamiltonian` are generated. So for 2 list-like parameters, the first elements in each are solved at the same time, then the second, etc Note that calling this function does not affect internal quantities directly, but flags them to be zipped at calculation time for relevant quantities.

Internally, adds the `label` value to the internal list of zipped parameter labels, and adds a flag in the form of `<label>:<parameter_name>` to each edge of the graph.

Parameters

- **parameters** (*dict*) – Parameter labels to scan together. Parameters are specified with a dictionary keyed by the either pair of states defining the coupling (e.g. `(0,1)`) or a previously specified label (e.g. `"probe"`) with items corresponding to the respective parameter name (e.g. `"detuning"`).
- **zip_label** (*optional, str*) – String label shorthand for the zipped parameters. The label for the axis of these parameters in `axis_labels()`. Does not affect functionality of the Sensor. If `None` (the default), the label used will be `"zip_" + <number>`, where <number> is the one index beyond the current length of the zip_parameters list.

Raises

- **RydiquleError** – If fewer than 2 labels are provided.
- **RydiquleError** – If any of the 2 labels are the same.
- **RydiquleError** – If the label contains the substring `"gamma"`, as this is used internally for decoherence matrix generation.
- **RydiquleError** – If any elements of `labels` are not labels of couplings in the sensor.
- **RydiquleError** – If any of the parameters specified by labels are already zipped.
- **RydiquleError** – If any of the parameters specified are not list-like.
- **RydiquleError** – If all list-like parameters are not the same length.

Notes

> **Note**
>
> This function should be called last after all Sensor couplings and dephasings have been added. Changing a coupling that has already been zipped removes it from the `self.zipped_parameters` list.

> **Note**
>
> Modifying the `Sensor._zip_labels` attribute directly can break some functionality and should be avoided. Use this function or *`unzip_parameters()`* instead.

> **Note**
>
> When defining the zip strings for states labelled with strings, be sure to additional `'` or `"` characters on either side of the labels, as demonstrated in the second example below.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label="probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"}, zip_label=
↪"detunings")
>>> print(s._zip_labels) #NOT modifying directly
['detunings']
>>> print(s.couplings.edges(data="detunings"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> print(s.get_hamiltonian().shape)#zipped parameters share an axis
(11, 3, 3)
```

Especially when states are labelled with tuples, specifying zips parameters with the states they couple can be cumbersome. In this case, it can be useful to either assign variables to the tuples defining the states, or to label the couplings.

```
>>> g = (0,0)
>>> e1, e2 = (1,-1), (1, 1)
>>> s = rq.Sensor([g, e1, e2])
>>> arr = np.linspace(-1,1,11)
>>> s.add_coupling((g,e1), detuning=arr, rabi_frequency=1, label="probe")
>>> s.add_coupling((e1,e2), detuning=arr, rabi_frequency=1, label="coupling")
>>> s.zip_parameters({((0,0),(1,-1)):"detuning", ((1,-1), (1, 1)):"detuning"},␣
↪zip_label="foo") #clunky
>>> print(s._zip_labels)
['foo']
>>> s.unzip_parameters("foo")
>>> s.zip_parameters({"probe":"detuning", "coupling":"detuning"}, zip_label=
↪"bar") #readable
>>> print(s._zip_labels)
['bar']
```

For maximum flexibility, any parameters specified as arrays with matching lengths can be zipped. This should be used with care, as some parameter combinations can be nonsensical.

```
>>> s = rq.Sensor(3)
>>> arr = np.linspace(-1,1,11)
>>> s.add_energy_shift(0, 0.5*arr)
>>> s.add_coupling(states=(0,1), detuning=arr, rabi_frequency=1, label="probe")
>>> s.add_coupling(states=(1,2), detuning=arr, rabi_frequency=1)
>>> s.add_decoherence((1,0), 0.1*arr)
>>> s.zip_parameters({(0,0):"e_shift", "probe":"detuning", (1,2):"detuning",␣
↪(1,0):"gamma"}, zip_label="foo")
>>> print(s._zip_labels) #NOT modifying directly
['foo']
>>> print(s.couplings.edges(data="foo"))
[(0, 0, 'e_shift'), (0, 1, 'detuning'), (1, 2, 'detuning'), (1, 0, 'gamma')]
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

zip_zips(**zip_labels: str*, *new_label: str | None = None*)

Combine multiple parameter zips into a single zip.

Given any number of labels of zips in the sensor, combines them so that they will all share a single axis in the stack. Note that this will override all previous zips in `zip_labels`, and they cannot be recovered.

Parameters

new_label (*string, optional*) – Label for the new zip that will replace the ones provided in `zip_labels`. If None, will be generated by joining all the strings of `zip_labels` with a "_" character, by default None.

Raises

- ***RydiquleError*** – If any of `zip_labels` do not exist in the `Sensor`.
- ***RydiquleError*** – If `new_label` contains a protected substring (such as "gamma").
- ***RydiquleError*** – If any of `zip_labels` are the same.
- ***RydiquleError*** – If any of the dimensions of the axes specified by `zip_labels` do not match.

Examples

```
>>> s = rq.Sensor(5)
>>> det = np.linspace(-1, 1, 11)
>>> s.add_coupling((0, [1,2]), rabi_frequency=1, detuning=det, label="foo")
>>> s.add_coupling((0, [3,4]), rabi_frequency=1, detuning=det, label="bar")
>>> print(s.get_hamiltonian().shape)
(11, 11, 5, 5)
>>> print(s.axis_labels())
['bar_detuning', 'foo_detuning']
>>> s.zip_zips("foo_detuning", "bar_detuning", new_label="foobar_detuning")
>>> print(s.get_hamiltonian().shape)
(11, 5, 5)
>>> print(s.axis_labels())
['foobar_detuning']
```

8.1.4 rydiqule.doppler_exact

Steady-state solver for analytical Doppler averaging

Code in this module implements analytical doppler averaging techniques described in Omar Nagib and Thad G. Walker, *Exact steady state of perturbed open quantum systems*, Phys. Rev. Research **7** 033076 (2025) https://doi.org/10.1103/kgsg-3npp

This solver computes the doppler averaged steady state of a sensor. In 1 spatial dimension problems, it averages analytically. In 2 or 3 spatial dimension problems, it averages one axis analytically and the remaining numerically.

Functions

solve_doppler_analytic(sensor[, ...])	Solves a sensor in steady state in the presence of doppler broadening, with one dimension analytically averaged.

rydiqule.doppler_exact.solve_doppler_analytic

rydiqule.doppler_exact.**solve_doppler_analytic**(*sensor: Sensor*, *doppler_mesh_method: UniformMethod | IsoPopMethod | SplitMethod | DirectMethod | None = None*, *analytic_axis: int | None = None*, *n_slices: int | None = None*, *rtol: float = 1e-05*, *atol: float = 1e-09*) → *Solution*

Solves a sensor in steady state in the presence of doppler broadening, with one dimension analytically averaged.

If the broadening is 1 dimensional, this function will solve analytically. If the broadening is 2 or 3 dimensional, this function will average analytically over the specified axis and numerically over the remaining axes.

This function uses the method outlined in Ref[1] for the analytic dimension.

This solver is considered more accurate than `solve_steady_state()` since it replaces direct sampling and solving of the velocity classes with a few tensor inversions and calculation of the numerical prefactor. This also leads to faster solves, approximately dictated by the ratio of samples along the doppler axis relative to the other parameter dimensions. Additionally, in sensors with 2 or 3 dimensional doppler broadening, this solver effectively reduces the dimension to 1 or 2, respectively, leading to faster solves.

If the sensor contains couplings with `time_dependence`, this solver will add those couplings at their $t = 0$ value to the steady-state hamiltonian to solve.

If insufficient system memory is available to solve the system in a single call, system is broken into "slices" of manageable memory footprint which are solved individually. This slicing behavior does not affect the result.

Parameters

- **sensor** (*Sensor*) – The sensor for which the solution will be calculated. It must define 1 and only 1 dimension of doppler shifts (ie one or more couplings with `kvec` with non-zero values on the same dimension).
- **doppler_mesh_method** (*dict, optional*) – If not `None`, should be a dictionary of meshing parameters to be passed to `doppler_classes()`. See `doppler_classes()` for more information on supported methods and arguments. If `None, uses the default doppler meshing. Default is `None.
- **analytic_axis** (*int, optional*) – Specifies over which axis the solver will average analytically. Defaults to the first nonzero axis.
- **n_slices** (*int or None, optional*) – How many sets of equations to break the full equations into. The actual number of slices will be the largest between this value and the minimum number of slices to solve the system without a memory error. If `None`, uses the minimum number of slices to solve the system without a memory error. Detailed information about slicing behavior can be found in `matrix_slice()`. Default is `None`.
- **rtol** (*float, optional*) – Relative tolerance parameter for checking 0-eigenvalues when calculating the doppler prefactor. Passed to `isclose()`. Defaults to 1e-5.
- **atol** (*float, optional*) – Absolute tolerance parameter for checking 0-eigenvalues when calculating the doppler prefactor. Passed to `isclose()`. Defaults to 1e-9.

[1] Omar Nagib and Thad G. Walker, Exact steady state of perturbed open quantum systems, Phys. Rev. Research **7** 033076 (2025) https://doi.org/10.1103/kgsg-3npp

Returns

An object containing the solution and related information.

Return type

Solution

References

8.1.5 rydiqule.doppler_utils

Utilities for implementing Doppler averaging

Functions

apply_doppler_weights(sols, velocities, volumes)	Calculates and applies the weight for each doppler class given unweighted solutions to doppler-shifted equations.
doppler_classes([method])	Defines which velocity classes to sample for doppler averaging.
doppler_mesh(doppler_velocities, spatial_dim)	Creates meshgrids of evaluation points and point "volumes" for doppler averaging.
gaussian3d(Vs)	Evaluate a multi-dimensional gaussian for the given detunings (in units of most probable speed).
generate_doppler_shift_eom(doppler_hamiltonia	Generates the EOMs for the supplied doppler shifts.
get_doppler_equations(base_eoms, ...[, ...])	Returns the equations for each slice of the doppler profile.

rydiqule.doppler_utils.apply_doppler_weights

rydiqule.doppler_utils.**apply_doppler_weights**(*sols: ndarray*, *velocities: ndarray*, *volumes: ndarray*) → ndarray

Calculates and applies the weight for each doppler class given unweighted solutions to doppler-shifted equations.

Works for both time-domain and stead-state solutions.

Parameters

- **sols** (*numpy.ndarray*) – The array of solutions over which to calculate weights.
- **velocities** (*numpy.ndarray*) – Array of shape `(n_dim, *n_dop)` where n_dim is the number of dimensions over which doppler shifts are being considered and `*n_dop` is a number of axes equal to n_dim with length equal to the number of doppler velocity classes which are being considered. The values correspond the velocity class in units of most probable speed.
- **volumes** (*numpy.ndarray*) – Array of shape equal to `velocities`. The values correspond to the spacings between doppler classes on each axis.

Returns

The weighted solution array of shape equal to that of `sols`.

Return type

numpy.ndarray

Raises

RydiquleError – If the shapes of `velocities` and `volumes` do not match.

rydiqule.doppler_utils.doppler_classes

`rydiqule.doppler_utils.doppler_classes`(*method: UniformMethod | IsoPopMethod | SplitMethod | DirectMethod | None = None*) → ndarray

Defines which velocity classes to sample for doppler averaging.

These are defined in units of the most probable speed of the Maxwell-Boltzmann distribution.

Note

To avoid issues, optical detunings should not leave densely sampled velocity classes. To avoid artifacts, the density of points should provide >~10 points over the narrowest absorptive feature. The default is a decent first guess, but for many problems the sampling mesh should be adjusted.

Parameters

method (*dict*) – Specifies method to use and any control parameters. Must contain the key `"method"` with one of the following options. Each method has suboptions that also need to be specified. Valid options are:

- `"uniform"`: Defines a uniformly spaced, dense grid. Configuration parameters include:
  - `"width_doppler"`: Float that specifies one-sided width of gaussian distribution to average over, in units of most probable speed. Defaults to 2.0.
  - `"n_uniform"`: Int that specifies how many points to use. Defaults to 1601.
- `"isopop"`: Defines a grid with uniform population in each interval. This method highly emphasizes physics happening near the 0 velocity class. If stuff is happening for non-zero velocity classes, it is likely to alias it unless `n_isopop` is large. See Ref[1] for details. Configuration parameters include:
  - `"n_isopop"`: Int that specifies how many points to use. Defaults to 400.
- `"split"`: Defines a grid with a dense central spacing and wide spacing wings. This method provides a decent compromise between uniform and isopop. It uses fewer points than uniform, but also works well for non-zero velocity class models (like Autler-Townes splittings). This is the default meshing method. Configuration parameters include:
  - `"width_doppler"`: Float that specifies one-sided width of coarse grided portion of the gaussian distribution. Units are in most probable speed. Defaults to 2.0.
  - `"width_coherent"`: Float that specifies one-sided width of fine grided portion of gaussian distribution. Units are in most probable speed. Defaults to 0.4.
  - `"n_doppler"`: Int that specifies how many points to use for the coarse grid. Note that points of the coarse grid that fall within the fine grid are dropped. Default is 201.
  - `"n_coherent"`: Int that specifies how many points to use for the fine grid. Default is 401.

  Note

  For the "split" method, a union of 2 samplings is taken, so the number of total points will not necessary be equal to the sum of `"n_coherent"` and `"n_doppler"`.

- `"direct"`: Use the supplied 1-D numpy array to build the mesh.
  - `"doppler_velocities"`: Mandatory parameter that holds the 1-D numpy array to use when building the mesh grids. Given in units of most probably speed.

[1] Andrew P. Rotunno, et. al. Inverse Transform Sampling for Efficient Doppler-Averaged Spectroscopy Simulation, AIP Advances 13, 075218 (2023) https://doi.org/10.1063/5.0157748

Returns
1-D array of velocities to be sampled.

Return type
numpy.ndarray

Examples

The defaults will sample more densely near the center of the distribution, (the "split" method) with a total of 561 classes.

```
>>> classes = rq.doppler_utils.doppler_classes() #use the default values
>>> print(classes.shape)
(561,)
```

Specifying "uniform" with no additional arguments produces 1601 evenly spaced classes by default.

```
>>> m = {"method":"uniform"}
>>> classes = rq.doppler_utils.doppler_classes(method=m)
>>> print(classes.shape)
(1601,)
```

Further specifying the number of points allows more dense or sparse sampling of the velocity distribution.

```
>>> m = {"method":"uniform", "n_uniform":801}
>>> classes = rq.doppler_utils.doppler_classes(method=m)
>>> print(classes.shape)
(801,)
```

The "split" method also has further specifications

```
>>> m = {"method":"split", "n_coherent":301, "n_doppler":501}
>>> classes = rq.doppler_utils.doppler_classes(method=m)
>>> print(classes.shape)
(701,)
```

References

rydiqule.doppler_utils.doppler_mesh

`rydiqule.doppler_utils.`**`doppler_mesh`**(*doppler_velocities: ndarray*, *spatial_dim: int*) → Tuple[ndarray, ndarray]

Creates meshgrids of evaluation points and point "volumes" for doppler averaging.

Parameters

- **`dop_velocities`** (*`numpy.ndarray`*) – A 1-D array of velocities to evaluate over. These should be normalized to the most probable velocity used by *`gaussian3d()`*.
- **`spatial_dim`** (*`int`*) – Number of spatial dimensions to grid over.

Returns

- **Vs** (*numpy.ndarray*) – Velocity evaluation points array of shape `(spatial_dim, spatial_dim*[len(dop_vel)])`.
- **Vols** (*numpy.ndarray*) – "Volume" of each mesh point. Has same shape as `Vs`.

Examples

```
>>> m = {"method":"uniform", "n_uniform":801}
>>> classes = rq.doppler_utils.doppler_classes(method=m)
>>> mesh, vols = rq.doppler_utils.doppler_mesh(classes, 2)
>>> print(type(mesh), type(vols))
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
>>> mesh_np = np.array(mesh)
>>> vols_np = np.array(vols)
>>> print(mesh_np.shape, vols_np.shape)
(2, 801, 801) (2, 801, 801)
```

rydiqule.doppler_utils.gaussian3d

`rydiqule.doppler_utils.`**`gaussian3d`**`(`*Vs: ndarray*`)` → ndarray

Evaluate a multi-dimensional gaussian for the given detunings (in units of most probable speed).

This is equivalent to a gaussian distribution with rms width $\sigma = 1/\sqrt{2}$.

Parameters

Vs (*numpy.ndarray*) – Array of normalized velocity classes for which to get the gaussian weighting.

Returns

Gaussian weights for the velocity classes. Has same shape as `Vs`.

Return type

numpy.ndarray

rydiqule.doppler_utils.generate_doppler_shift_eom

`rydiqule.doppler_utils.`**`generate_doppler_shift_eom`**`(`*doppler_hamiltonians: ndarray*, *ground_removed: bool = True*`)` → ndarray

Generates the EOMs for the supplied doppler shifts.

Multiply the output by the velocity in each dimension, then add to the normal EOMs to get the full Doppler shifted EOMs.

Parameters

- **doppler_hamiltonians** (*numpy.ndarray*) – Hamiltonians of only the doppler shifts, one for each spatial dimension to be averaged over.
- **ground_removed** (*bool, optional*) – Whether to remove the ground state from the equations. Default is True.

Returns

Corresponding LHS EOMs with ground removed and in the real basis.

Return type

numpy.ndarray

rydiqule.doppler_utils.get_doppler_equations

`rydiqule.doppler_utils.`**`get_doppler_equations`**`(`*base_eoms: ndarray*, *doppler_hamiltonians: ndarray*, *Vs: ndarray*, *ground_removed: bool = True*`)` → ndarray

Returns the equations for each slice of the doppler profile.

A new axes corresponding to these slices are appended to the beginning. For example, if equations are of shape `(m,m)` and there are `n_doppler` doppler values being sampled, the return will be of shape `(n_doppler, m, m)`.

Parameters

- **base_eoms** (*numpy.ndarray*) – Stacked square arrays representing the unshifted equations, i.e. the theoretical equations for an ensemble of atoms with zero momentum.
- **doppler_hamiltonians** (*numpy.ndarray*) – Arrays of hamiltonians with only doppler shifts present. One for each spatial dimension needed. See `get_doppler_shifts()` for details.
- **Vs** (*numpy.ndarray*) – Mesh of velocity classes to sample, with same spatial dimensions as `dop_ham`. See `doppler_mesh()` for details.
- **ground_removed** (*bool, optional*) – Whether to remove the ground state from the equations. Default is True.

Returns

An array of shape `(*Vs.shape[1:], *base_eoms.shape)` which is a, potentially multi-dimensional, stack of individual equations of shape `(m, m)`. Each slice of this stack is an equation of shape `(m, m)` with the corresponding doppler shifts applied.

Return type

numpy.ndarray

Note

Each doppler shift is equal to `k_i*vP*det_i`, in units of Mrad/s, where `i` denotes the einstein summation along the spatial dimensions. `det` is the normalized velocity class, with `vP*det_i=v_i` giving the velocity. `vP` is the most probable speed from the Maxwell-Boltzmann distribution: sqrt(2*kB*T/m). `k_i` is the k-vector of the field along the same axis as `det_i`. `doppler_hamiltonians` provides `k_i*vP`, `Vs` provides `det_i`.

Classes

DirectMethod
IsoPopMethod
SplitMethod
UniformMethod

rydiqule.doppler_utils.DirectMethod

class rydiqule.doppler_utils.**DirectMethod**

Bases: `TypedDict`

__init__(**args*, ***kwargs*)

Methods

__init__(*args, **kwargs)	
clear()	
copy()	
fromkeys(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
get(key[, default])	Return the value for key if key is in the dictionary, else default.
items()	
keys()	
pop(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.

continues on next page

Table 8.19 – continued from previous page

popitem()	Remove and return a (key, value) pair as a 2-tuple.
setdefault(key[, default])	Insert key with a value of default if key is not in the dictionary.
update([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
values()	

Attributes

method
doppler_velocities

clear() → None. Remove all items from D.

copy() → a shallow copy of D

doppler_velocities: ndarray | Sequence

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Literal['direct']

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

rydiqule.doppler_utils.IsoPopMethod

class rydiqule.doppler_utils.**IsoPopMethod**

Bases: TypedDict

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`clear`()	
`copy`()	
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(key[, default])	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
`popitem`()	Remove and return a (key, value) pair as a 2-tuple.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`method`
`n_isopop`

clear() → None. Remove all items from D.

copy() → a shallow copy of D

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Required[Literal['isopop']]

n_isopop: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([E,]**F) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

rydiqule.doppler_utils.SplitMethod

class rydiqule.doppler_utils.**SplitMethod**

Bases: `TypedDict`

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`clear`()	
`copy`()	
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(key[, default])	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
`popitem`()	Remove and return a (key, value) pair as a 2-tuple.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]
`values`()	

Attributes

`method`
`width_doppler`
`n_doppler`
`width_coherent`
`n_coherent`

clear() → None. Remove all items from D.

copy() → a shallow copy of D

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Required[Literal['split']]

n_coherent: int

n_doppler: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, /)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

width_coherent: float

width_doppler: float

rydiqule.doppler_utils.UniformMethod

class rydiqule.doppler_utils.**UniformMethod**

Bases: TypedDict

__init__(**args*, ***kwargs*)

Methods

`__init__`(*args, **kwargs)	
`clear`()	
`copy`()	
`fromkeys`(iterable[, value])	Create a new dictionary with keys from iterable and values set to value.
`get`(key[, default])	Return the value for key if key is in the dictionary, else default.
`items`()	
`keys`()	
`pop`(k[,d])	If the key is not found, return the default if given; otherwise, raise a KeyError.
`popitem`()	Remove and return a (key, value) pair as a 2-tuple.
`setdefault`(key[, default])	Insert key with a value of default if key is not in the dictionary.
`update`([E,]**F)	If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

continues on next page

Table 8.25 – continued from previous page

values()

Attributes

method
width_doppler
n_uniform

clear() → None. Remove all items from D.

copy() → a shallow copy of D

classmethod fromkeys(*iterable*, *value=None*, */*)

Create a new dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

method: Required[Literal['uniform']]

n_uniform: int

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem()

Remove and return a (key, value) pair as a 2-tuple.

Pairs are returned in LIFO (last-in, first-out) order. Raises KeyError if the dict is empty.

setdefault(*key*, *default=None*, */*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update([*E*,]***F*) → None. Update D from dict/iterable E and F.

If E is present and has a .keys() method, then does: for k in E: D[k] = E[k] If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v In either case, this is followed by: for k in F: D[k] = F[k]

values() → an object providing a view on D's values

width_doppler: float

8.1.6 rydiqule.exceptions

Rydiqule Custom Exceptions and Warnings

This module defines a number of custom Errors and Warnings for use in Rydiqule.

It ensures that all RydiquleWarnings are always shown. This behavior can be changed by calling `warnings.simplefilter()`.

It also defines custom exception handlers to hide the raise statement associated with RydiquleErrors. This behavior can be suppressed by calling `rq.debug_state(True)`.

Functions

`debug_state`()	Returns current rydiqule debug state
`quiet_exception_handler`(exception_type, ...)	
`set_debug_state`(state)	Controls DEBUG state of rydiqule.

rydiqule.exceptions.debug_state

rydiqule.exceptions.**debug_state**()

Returns current rydiqule debug state

Returns
Current debug state

Return type
bool

rydiqule.exceptions.quiet_exception_handler

rydiqule.exceptions.**quiet_exception_handler**(*exception_type*, *exception*, *tb*)

rydiqule.exceptions.set_debug_state

rydiqule.exceptions.**set_debug_state**(*state: bool*)

Controls DEBUG state of rydiqule.

Parameters
state (*bool*) – If True, full error tracebacks for RydiquleErrors will be shown. If False (default behavior), final raise statement is suppressed in the traceback.

Exceptions

`AtomError`	An error in interacting with ARC
`CouplingNotAllowedError`	Indicated coupling is not allowed (eg dipole-forbidden)
`NLJMWarning`	Indicates `Cell` has likely been called with the old state specification [n, l, j, m] for a solve that did not intend to use fine-structure magnetic sublevels.
`PopulationNotConservedWarning`	Indicates population will not be conserved in the model.
`RWAWarning`	Indicates the coupling is using a large transition frequency outside the rotating wave approximation.
`RydiquleError`	A *rydiqule* error
`RydiquleWarning`	Indicates a rydiqule-specfic warning.
`TimeDependenceWarning`	Indicates a time-dependent coupling is being used in a steady-state context

rydiqule.exceptions.AtomError

exception rydiqule.exceptions.**AtomError**

An error in interacting with ARC

rydiqule.exceptions.CouplingNotAllowedError

exception rydiqule.exceptions.**CouplingNotAllowedError**

Indicated coupling is not allowed (eg dipole-forbidden)

rydiqule.exceptions.NLJMWarning

exception `rydiqule.exceptions.`**`NLJMWarning`**

Indicates `Cell` has likely been called with the old state specification [n, l, j, m] for a solve that did not intend to use fine-structure magnetic sublevels.

rydiqule.exceptions.PopulationNotConservedWarning

exception `rydiqule.exceptions.`**`PopulationNotConservedWarning`**

Indicates population will not be conserved in the model.

rydiqule.exceptions.RWAWarning

exception `rydiqule.exceptions.`**`RWAWarning`**

Indicates the coupling is using a large transition frequency outside the rotating wave approximation.

rydiqule.exceptions.RydiquleError

exception `rydiqule.exceptions.`**`RydiquleError`**

A *rydiqule* error

Indicates a Rydiqule-specific error. It is a thin wrapper around Exception.

rydiqule.exceptions.RydiquleWarning

exception `rydiqule.exceptions.`**`RydiquleWarning`**

Indicates a rydiqule-specfic warning.

All other rydiqule warnings derive from this class.

Disable globally by calling `warnings.simplefilter('ignore', rq.RydiquleWarning)` or locally using the `warnings.catch_warnings` context manager.

rydiqule.exceptions.TimeDependenceWarning

exception `rydiqule.exceptions.`**`TimeDependenceWarning`**

Indicates a time-dependent coupling is being used in a steady-state context

8.1.7 rydiqule.experiments

Standard methods for converting results to physical values.

Functions

get_snr(sensor, param_label[, ...])	Calculate a Sensor's signal-to-noise ratio in standard deviation, in a 1Hz bandwidth, to a specified signal parameter, assuming a homodyne measurement of optical field.

rydiqule.experiments.get_snr

`rydiqule.experiments.`**`get_snr`**(*sensor: Sensor, param_label: str, phase_quadrature: bool = False, diff_nearest: bool = False, **kwargs*) → Tuple[ndarray, List[ndarray]]

Calculate a Sensor's signal-to-noise ratio in standard deviation, in a 1Hz bandwidth, to a specified signal parameter, assuming a homodyne measurement of optical field.

SNR is calculated with respect to the signal parameter, relative to the initial value of the signal parameter. The returned mesh is similarly transformed from the typical sensor mesh, by replacing the total value of the signal parameter with the deviation in the signal parameter.

The conventions used follow that of[1].

Note

The default is to return the SNR of an amplitude quadrature measurement. To convert to a power measurement (i.e. Ω_p^2), the amplitude quadrature SNR must be divided by 2. To get the SNR in variance, square the result.

Parameters

- **sensor** (Sensor) – sensor for which SNR should be calculated. The definition of sensor.couplings should contain at least one coupling with a list-like parameter. For the list-like parameter, the first array element is the "base" against which SNR for each other value is calculated.
- **param_label** (*str*) – Label of the axis with respect to which SNR is calculated. See `Sensor.axis_labels()` for more details on axis labeling. The value corresponding to this label should be the list-like parameter with respect to which SNR should be calculated. This parameter list must have at least two elements, and SNR is calculated relative to the first element in the list for all other elements in the list.
- **phase_quadrature** (bool, optional) – Whether the sensor is measured in the phase quadrature of the probe laser. False denotes measurement in the amplitude quadrature. Default is False.
- **diff_nearest** (*bool, optional*) – Controls method by which the SNR is calculated. The default (False) calculates the SNR with respect to the 0 index value. Setting True calculates the SNR with respect to nearest neighbor differences.
- **kwargs** (*dict, optional*) – Additional keyword arguments to pass to `rq.solve_steady_state()`.

Returns

- **snrs** (*numpy.ndarray*) – Array of SNRs for the sensor with respect to the change in the signal parameter. Calculated in units of amplitude relative to noise standard deviation. SNR referenced to 1 second BW.
- **mesh** (*tuple(numpy.ndarray)*) – Numpy meshgrid of the coupling parameters that yield each snr. The signal parameter axis now shows the signal change.

Raises

RydiquleError – If the specified param_label is not in `Sensor.axis_labels()`

Examples

```
>>> atom = "Rb85"
>>> [g,e] = rq.D2_states("Rb85")
>>> c = rq.Cell('Rb85', [g,e], cell_length=0.0001)
>>> c.add_coupling(states=(g,e), rabi_frequency=np.linspace(1e-6, 1, 5),␣
↪detuning=1, label="probe")
>>> snr, mesh = rq.get_snr(c, 'probe_rabi_frequency')
>>> print(snr)
[        0.         13654034.1 27301261.5 40934886.9
 54548137.6]
>>> print(mesh)
[array([0.       , 0.25     , 0.499999, 0.749999, 0.999999])]
```

[1] D. H. Meyer, C. O'Brien, D. P. Fahey, K. C. Cox, and P. D. Kunz, "Optimal atomic quantum sensing using electromagnetically-induced-transparency readout," Phys. Rev. A, vol. 104, p. 043103, 2021.

References

8.1.8 rydiqule.rydiqule_utils

General rydiqule package utilities

Functions

about([obscure_paths, show_numpy_config])	About box describing Rydiqule and its core dependencies.

rydiqule.rydiqule_utils.about

rydiqule.rydiqule_utils.**about**(*obscure_paths: bool = True*, *show_numpy_config: bool = True*)

About box describing Rydiqule and its core dependencies.

Prints human readable strings of information about the system.

Parameters

- **obscure_paths** (*bool, optional*) – Remove user directory from printed paths. Default is True.
- **show_numpy_config** (*bool, optional*) – Show the numpy config for BLAS/LAPACK backends. Default is True.

Examples

```
>>> rq.about()

        Rydiqule
    ================

Rydiqule Version:     2.1.1
Installation Path:    ~\rydiqule\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.2.5
SciPy Version:        1.16.0
Matplotlib Version:   3.10.0
ARC Version:          3.9.0
Python Version:       3.11.10
Python Install Path:  ~\miniconda3\envs\arc
Platform Info:        Windows (AMD64)
CPU Count:            16 @ 3.91 GHz
Total System Memory:  256 GB

     NumPy backends
    ================

blas: provided by mkl-sdl (2023.1)
lapack: provided by mkl-sdl (2023.1)
```

8.1.9 rydiqule.sensor

Sensor objects that control solvers.

Module Attributes

BASE_SCANNABLE_KEYS	Reference list of all coherent coupling keys that support rydiqule's stacking convention.
BASE_EDGE_KEYS	Reference list of all keys that can be specified with values in a coherent coupling.

rydiqule.sensor.BASE_SCANNABLE_KEYS

rydiqule.sensor.**BASE_SCANNABLE_KEYS = ['detuning', 'rabi_frequency', 'phase', 'e_shift']**

Reference list of all coherent coupling keys that support rydiqule's stacking convention. Note that all decoherence keys (keys beginning with `gamma_`) are supported, but handled separately.

rydiqule.sensor.BASE_EDGE_KEYS

rydiqule.sensor.**BASE_EDGE_KEYS = ['states', 'detuning', 'rabi_frequency', 'transition_frequency', 'phase', 'kvec', 'time_dependence', 'label', 'dipole_moment', 'coherent_cc']**

Reference list of all keys that can be specified with values in a coherent coupling. Subclasses which inherit from *Sensor* should override the `valid_parameters` attribute, NOT this list. The `valid_parameters` attribute is initialized as a copy of `BASE_EDGE_KEYS`.

Classes

Sensor(states, *couplings[, vP])	Class that contains minimum information necessary to run the solvers.

rydiqule.sensor.Sensor

class rydiqule.sensor.**Sensor**(*states: int | Sequence[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], *couplings: Dict, vP: float | None = None*)

Bases: `object`

Class that contains minimum information necessary to run the solvers.

Consider this class the theorist's interface to the solvers. It requires nearly complete, explicit specification of inputs. This allows for very fine control of the solvers, including the ability to solve systems that are not entirely physical.

__init__(*states: int | Sequence[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], *couplings: Dict, vP: float | None = None*) → None

Initializes the Sensor with the specified basis .

Can be specified as either an integer number of states (which will automatically label the states `[0,...,basis_size]`) or list of valid state specifications.

Parameters

- **states** (*int or list of statespec*) – The specification of the basis size and labelling for a new `Sensor`. Can be either a integer or a list of valid state specifications. If specified as an integer `n`, the created `Sensor` will have `n` states labelled as `0,...n`. Valid state specifications are tuples containing either numbers or strings, optionally a list

of the same. In the case of a list, the tuple will be converted into a list of tuples with each corresponding to one element in the list element element of the specification. See the `Examples` section for examples on how to specify groups of states. Note that with only a single statespec, it must be passed as an element of a list

- ***couplings** (*tuple(dict)*) – Couplings dictionaries to pass to *add_couplings()* on sensor construction.
- **vP** (*float, optional*) – Most probable speed of the 3D Maxwell-Boltzmann distribution of the ensemble. Calculated as $\sqrt{2kT/m}$ and is provided in units of m/s. This parameter is only necessary to perform Doppler-broadened solves.

Raises

- ***RydiquleError*** – If `basis` is not an integer or iterable.
- ***RydiquleError*** – If any of the state label specifications of basis are the wrong type.

Examples

Providing an integer will define a sensor with the given basis size, labelled with ascending integers.

```
>>> s = rq.Sensor(3)
>>> s.states
[0, 1, 2]
```

States can also be defined with a list of integers:

```
>>> s = rq.Sensor([0, 1, 2])
>>> s.states
[0, 1, 2]
```

States can also be strings

```
>>> s = rq.Sensor([0, 'e1', 'e2'])
>>> s.states
[0, 'e1', 'e2']
```

States can be defined with tuples. These can be thought of as quantum numbers, although no physics around quantum numbers exist in `Sensor`, so the values are completely general.

```
>>> s = rq.Sensor([(1,-1),(1,1)])
>>> s.states
[(1, -1), (1, 1)]
```

States can be specified in groups with a "state specification", which will expand lists of quantum numbers

```
>>> statespec = (1,[-1,0,1])
>>> s = rq.Sensor([(0,0), statespec])
>>> s.states
[(0, 0), (1, -1), (1, 0), (1, 1)]
```

Methods

__init__(states, *couplings[, vP])	Initializes the Sensor with the specified basis .
add_coupling(states, **kwargs)	Add a coupling between states or groups of states.
add_coupling_group(states1, states2, label)	Adds a group of couplings to a Sensor.
add_couplings(*couplings, **extra_kwargs)	Add any number of couplings between pairs of states.

continues on next page

Table 8.33 – continued from previous page

add_decoherence(statespecs, gamma, **kwargs)	Add a coupling between states or groups of states.
add_decoherence_group(states1, states2, ...)	Adds a group of decoherences to the Sensor.
add_energy_shift(statespec, shift, **kwargs)	Add an energy shift to a single state or a group of states.
add_energy_shift_group(states, shift[, ...])	Add energy shifts to a group of states, optionally with a modifying prefactor for each.
add_energy_shifts(shifts)	Wrapper for `Sensor.add_energy shift b()`.
add_self_broadening(state, gamma[, label, ...])	Specify self-broadening (such as collisional broadening) of a level.
add_self_broadening_group(states, gamma[, ...])	Specify self-broadening (such as collisional broadening) of a group of states.
add_single_coupling(states[, ...])	Adds a single coupling of states to the system.
add_single_decoherence(states, gamma[, ...])	Add decoherent coupling to the graph between two states.
add_single_energy_shift(state, shift[, label])	Add an energy shift to a state.
add_transit_broadening(gamma_transit[, ...])	Adds transit broadening by adding a decoherence from each node to ground.
axis_labels()	Get a list of axis labels for stacked hamiltonians.
coupling_subgraph(coupling)	Returns a subgraph view of the couplings graph corresponding to `coupling`.
couplings_with(*keys[, method])	Returns a version of self.couplings with only the keys specified.
decoherence_matrix()	Build a decoherence matrix out of the decoherence terms of the graph.
dm_basis()	Generate basis labels of density matrix components.
get_couplings()	Returns the couplings of the system as a dictionary
get_doppler_shifts()	Returns the Hamiltonian with only detunings set to the most probable doppler shift values for each spatial dimension.
get_hamiltonian()	Creates the Hamiltonians from the couplings defined by the fields.
get_hamiltonian_diagonal(values[, no_stack])	Apply addition and subtraction logic corresponding to the direction of the couplings.
get_parameter_mesh()	Returns the parameter mesh of the sensor.
get_rotating_frames()	Determines the rotating frames for the disconnected subgraphs.
get_time_dependence()	Function which returns a list of the `time_dependence` functions.
get_time_hamiltonian(t)	Get the system hamiltonian at a specific time, t.
get_time_hamiltonian_components()	Get time-dependent components of the hamiltonian.
get_transition_frequencies()	Gets an array of the diagonal elements of the Hamiltonian from the field detunings.
get_value_dictionary(key)	Get subset of dictionary coupling parameters.
group_variable_parameters([apply_mesh])	
int_states_map([invert])	Get a dictionary mapping between state labels and their corresponding integer ordering.
set_experiment_values(probe_freq, kappa[, ...])	Sets attributes needed for observable calculations.
set_gamma_matrix(gamma_matrix)	Set the decoherence matrix for the system.
spatial_dim()	Returns the number of spatial dimensions doppler averaging will occur over.
states_with_spec(statespec)	Return a list of all states in the sensor matching the `state_spec` pattern.
unzip_parameters(zip_label[, verbose])	Remove a set of zipped parameters from the internal zip_labels list.

continues on next page

Table 8.33 – continued from previous page

variable_parameter_sort(par)	Assistance function which determines the sorting order of elements parameters in sensor.
variable_parameters([apply_mesh])	Property to retrieve the values of parameters that were stored on the graph as arrays.
zip_parameters(parameters[, zip_label])	Define 2 scannable parameters as "zipped" so they are scanned in parallel.
zip_zips(*zip_labels[, new_label])	Combine multiple parameter zips into a single zip.

Attributes

atom_mass	Mass of an atom in the vapor cell, in kilograms.
basis_size	Property to return the number of nodes on the Sensor graph.
beam_area	Cross-sectional area of the probing beam, in square meters.
cell_length	Optical path length of the medium, in meters.
eta	Noise density prefactor, in units of root(Hz).
kappa	Differential prefactor, in units of (rad/s)/m.
probe_freq	Probing transition frequency, in rad/s.
probe_tuple	Coupling edge that corresponds to the probing field.
states	Property which gets a list of labels for the sensor in the order defined in `__init__()`.
temp	Temperature of the vapor cell, in Kelvin.
vP	Most probable speed of the 3D Maxwell-Boltzmann distribution.
v_th	Thermal velocity of the atoms in vapor cell, in meters per second.

__sort_couplings(*coupling*)

Helper function for __str__ to sort couplings by states

_add_coherent_data(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], **field_params*) → None

Function for internal use which will ensure the supplied couplings is valid, add the field to self.couplings.

Exists to abstract away some of the internally necessary bookkeeping functionality from user-facing classes.

Parameters

- **states** (*tuple*) – The integer pair of states to be coupled.
- ****field_params** (*dict*) – The dictionary of couplings parameters. For details on the keys of the dictionary see `add_coupling()`.

_coupling_with_label(*label: str | Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]*) → Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]

Helper function to return the pair of states corresponding to a particular label string. For internal use.

_expand_dims()

Converts the 1-D arrays in the sensor into shapes that allows for rydiqule stacking.

_probe_tuple: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]] | None = None

_remove_edge_data(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], kind: str*)

Helper function to remove all data that was added with a `add_coupling()` call or `add_decoherence()` call. Needed to ensure that two nodes do not have coherent couplings pointing both ways and to invalidate existing zip parameter couplings.

Parameters

- **states** (`tuple`) – Edge from which to remove data.
- **kind** (`str`) – What type of data to remove. Valid options are `coherent` coherent couplings or the incoherent key to be cleared (must start with `gamma`).

Raises

RydiquleError – If `kind` is not `'coherent'` and doesn't begin with `'gamma'`

_stack_shape(*time_dependence: Literal['steady', 'time', 'all'] = 'all'*) → Tuple[int, ...]

Internal function to get the shape of the tuple preceding the two hamiltonian axes in `get_hamiltonian()`

_states_valid(*states: Sequence*) → Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]

Confirms that the provided states are in a valid format.

Typically used internally to validate states added. If provided as a form other than a tuple, first casts to a tuple for consistent indexing.

Checks that `states` contains 2 elements, can be interpreted as a tuple, and that both states lie inside the basis.

Parameters

states (`iterable`) – iterable of to validate. Should be a pair of integers that can be cast to a tuple.

Returns

Length 2 tuple of validated state labels.

Return type

tuple

Raises

- **RydiquleError** – If `states` has more than two elements.
- **TypeError** – If `states` cannot be converted to a tuple.
- **RydiquleError** – If either state in `states` is outside the basis.

_vP: float | None = None

add_coupling(*states: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], **kwargs*)

Add a coupling between states or groups of states.

Wraps the `add_single_coupling()` and `add_coupling_group()` functions, and dispatches to the appropriate one depending on the number of states in the `states` argument. Additional keyword arguments will be passed unmodified to the relevant method. See documentation of those functions for details on keyword argument options.

If each state specification in `states` correspond to a single state, the corresponding states will be passed to `add_single_coupling()`. If either or both specifications correspond to multiple states, the corresponding lists will be passed as the `states1` and `states2` lists in `add_coupling_group()`.

If this is the first time `add_coupling` has been called for this `Sensor`, sets the `probe_tuple` attribute to the `states` specification , which is used as the default, for calculating observable values, in a `Solution` after solving. For this reason, this function is preferred over `add_single_coupling()` and `add_coupling_group()` outside of special circumstances. If couplings are added with either of the specific dispatched functions, `probe_tuple` should be set manually.

Parameters

- **states** (*tuple of Statespecs*) – The states or state manifolds of the coupling. If both are integers or state specifications matching a single state in the `Sensor`, `add_single_coupling()` is dispatched. If either argument is a string pattern matching multiple states, `add_coupling_group()` is dispatched.
- ****kwargs** – Additional keyword arguments passed to the relevant function. See the documentation for `add_single_coupling()` and `add_coupling_group()` for details on valid keyword arguments.

Notes

..note:
Outside of specific use cases for users well-versed in the `rydiqule` code base, this method is preferred over `add_single_coupling()` and `add_coupling_group()` since it appropriately handles necessary backend bookkeeping.

Examples

Couplings are added identically regardless of how states are labelled.

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

```
>>> s = rq.Sensor(['g','e'])
>>> s.add_coupling(('g','e'), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

Couplings can have list-like parameters, in which case the resulting rydiqule will compute hamiltonians for all values simultaneously. Here 101 x 21 = 2,121 2x2 Hamiltonians are generated simultaneously, with one for every combination of parameters, and arranged into a single array.

```
>>> s = rq.Sensor(2)
>>> det=np.linspace(-10, 10, 101)
>>> rabi = np.linspace(-1, 1, 21)
>>> s.add_coupling((0,1), detuning=det, rabi_frequency=rabi, label="laser")
>>> print(s.get_hamiltonian().shape)
(101, 21, 2, 2)
```

Couplings can be be defined between manifolds of states with state specifications. The values for rabi frequencies of individual states are modified by the `coupling_coefficients` keyword argument. To avoid cumbersome numbers of nested brackets, it is advisable to name manifolds with variables. Note that `StateSpec`s can be expanded with :func:`~.sensor_utils.expand_statespec` for this purpose.

```
>>> g = (0,0) #statespec for ground
>>> excited = (1,[-1,0,1]) #statespec for excited
>>> [e1,e2,e3] = rq.sensor_utils.expand_statespec(excited)
>>> cc = {
...     (g, e1): 0.25,
...     (g, e2): 0.5,
...     (g, e3): 0.25,
... } # coupling coefficients
```

(continues on next page)

(continued from previous page)

```
>>> s = rq.Sensor([g, excited])
>>> s.add_coupling((g, excited), rabi_frequency=10, detuning=1, coupling_
↪coefficients=cc, label="laser")
>>> print(s.get_hamiltonian())
[[ 0.   +0.j  1.25+0.j  2.5 +0.j  1.25+0.j]
 [ 1.25-0.j -1.  +0.j  0.  +0.j  0.  +0.j]
 [ 2.5 -0.j  0.  +0.j -1.  +0.j  0.  +0.j]
 [ 1.25-0.j  0.  +0.j  0.  +0.j -1.  +0.j]]
```

This function sets the `probe_tuple` for the first call, but not subsequent calls. This makes it preferred over `add_single_coupling()` and `add_coupling_group()`, which do not have this behavior.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> e2 = (2,0)
>>> s = rq.Sensor([g, e1, e2])
>>> print(s.probe_tuple)
None
>>> s.add_coupling((g, e1), rabi_frequency=10, detuning=1, label="red")
>>> print(s.probe_tuple)
((0, 0), (1, [-1, 0, 1]))
>>> s.add_coupling((e1,e2), rabi_frequency=1, detuning=2, label="blue")
>>> print(s.probe_tuple)
((0, 0), (1, [-1, 0, 1]))
```

For state manifolds, list-like parameters are automatically zipped. See `Sensor.zip_parameters()` for more details on the mechanics of zipping parameters.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> s = rq.Sensor([g, e1])
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((g, e1), rabi_frequency=10, detuning=det, label="red")
>>> print(s.couplings.edges)
[((0, 0), (1, -1)), ((0, 0), (1, 0)), ((0, 0), (1, 1))]
>>> print(s._zip_labels)
['red_detuning']
>>> print(s.get_hamiltonian().shape)
(11, 4, 4)
```

add_coupling_group(*states1: List[int | str | Tuple[float, ...]], states2: List[int | str | Tuple[float, ...]], label: str, rabi_frequency: float | List[float] | ndarray | None = None, detuning: float | List[float] | ndarray | None = None, transition_frequency: float | None = None, coupling_coefficients: dict | None = None, time_dependence: Callable | Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Callable | None] | None = None, **kwargs*)

Adds a group of couplings to a Sensor.

Given 2 lists of states, iterates over each combination of states in the two lists, add performs the `add_single_coupling()` on that pair of states. All additional parameters are passed directly to the `add_single_coupling` function.

Additionally, a multiplicative factor can be applied to the rabi frequency of each coupling (i.e. Clebsch-Gordon coefficients). These factors are provided by the `` cc` (coupling coefficient) parameter as a dictionary with keys corresponding to state pairs in the groups, and the value being the multiplicative factor applied to `rabi_frequency `` when the Hamiltonian is generated. Note that these `cc` values cannot be arrays. The corollary for state energy (e.g. for `detuning` or `transition_frequency`) is handled via `add_energy_shifts()`. If no dic-

tionary is supplied for for `coherent_cc`, *all* coupling coefficients are set to 1.0, effectively meaning that the base `rabi_frequency` supplied is what is added to the Hamiltonian. If a dictionary is supplied, any couplings whose coefficient is not specified by the dictionary will be left off the graph.

If any of the parameters are specified as arrays, the associated couplings will be applied to all couplings and automatically zipped together with the label specified by `label`. For the purposes of axis labelling, the parameters will be zipped in the order`rabi_frequency`, `detuning`, `transition_frequency`.

Note that unlike *add_coupling()*, this function does not set the `probe_tuple` attribute, so if used to add the first coupling, `probe_tuple` must be set manually.

Parameters

- **states1** (*list[str or int]*) – List of states in the lower energy group of states. Must be integers or string values which correspond to states in the Sensor.
- **states2** (*list[str or int]*) – List of states in the higher energy group of states. Must be integers or string values which correspond to states in the Sensor.
- **label** (*str*) – Required string label denoting what the group of couplings is called. Used to apply a label in *zip_parameters()*.
- **rabi_frequency** (*ScannableParameter, optional*) – Floating point value or list of values for the base rabi frequency of the coupling group. Multiplied by values specified in `cc` for individual couplings, often accounting for variations in dipole moment. Default is None.
- **detuning** (*ScannableParameter, optional*) – Base detuning for the coupling group. If specified, every coupling in the group will be treated in the rotating frame. Can be modified through energy level shifts on individual states specified by *add_energy_shift()*. Default is None.
- **transition_frequency** (*float, optional*) – Base transition frequency for the coupling group. Individual states can be shifted via *add_energy_shift()*. Default is None.
- **coupling_coefficients** (*dict, optional*) – Individual coupling coefficients passed to the *add_single_coupling()* method. If provided, defined by a dictionary keyed with tuples of states corresponding to couplings in this group, with values equal to the coupling coefficient to be passed to the `add_single_coupling` call for that coupling. If any entries are absent in the provided dictionary, they are assumed to not be coupled, and no coupling will be added for that transition. If `None`, defaults to a dictionary containing every coupling in coupling in the group with `None` for all values (defaulting to 1.0 when passed to `add_single coupling`). Defaults to `None`.
- **time_dependence** (*scalar function or dict of scalar functions, optional*) – Time-dependent scalar factor that is multiplied by the rabi frequency in Hamiltonian generation. Can be specified as a single function, in which case the function will be used as the `time_dependence` argument for each coupling in the group (see *add_single_coupling()*), Can also be specified as a dictionary mapping state pairs in the coupling to individual functions which will be applied to the associated coupling in the same manner. In the case of a dictionary specification, each unspecified coupling will default to `time_dependence=None`.

Raises

RydiquleError – If `states1` and `states2` only have one state. Use *add_coupling()* instead.

Note

Note

The *add_coupling()* is typically preferred over this method, since it allows for shorthand specification of groups, and sets the *Sensor.probe_tuple* attribute.

Note

If a *CouplingNotAllowedError* is raised while adding the individual couplings for the group, couplings that raised the error will be ignored.

Examples

Energy shifts added to remove degenerate energy levels. If no clebsch-gordon coefficients are supplied, ALL default to 1

```
>>> s = rq.Sensor(['a1', 'a2', 'b1', 'b2'])
>>> s.add_energy_shifts({'a2':0.1, 'b2':0.1})
>>> s.add_coupling_group(['a1','a2'], ['b1','b2'], detuning=1, rabi_
↪frequency=1, label='example')
>>> s.get_hamiltonian()
array([[ 0. +0.j,  0. +0.j,  0.5+0.j,  0.5+0.j],
       [ 0. +0.j,  0.1+0.j,  0.5+0.j,  0.5+0.j],
       [ 0.5-0.j,  0.5-0.j, -1. +0.j,  0. +0.j],
       [ 0.5-0.j,  0.5-0.j,  0. +0.j, -0.9+0.j]])
```

If the cc dictionary is specified, any unspecified terms are skipped on the graph. Note that although `(0,3)` is in the coupling group, it is omitted from the graph since it is not in `coupling_coefficients`.

```
>>> s = rq.Sensor(4)
>>> cc = {(0,1):0.5, (0,2):0.5}
>>> s.add_coupling_group([0],[1,2,3], detuning=1, rabi_frequency=1,␣
↪coupling_coefficients=cc, label='foo')
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 4 states and 2 coherent␣
↪couplings.
States: [0, 1, 2, 3]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_0, coherent_cc: 0.5}
    (0,2): {rabi_frequency: 1, detuning: 1, phase: 0, kvec: (0, 0, 0),␣
↪label: foo_1, coherent_cc: 0.5}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

For list-like parameters, the couplings are treated as originating from a single laser and that parameter is zipped across all couplings in the group.

```
>>> g = (0,0) #statespec for ground
>>> e1 = (1,[-1,0,1]) #statespec for 1st excited
>>> s = rq.Sensor([g, e1])
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling_group([(0,0)], [(1,-1), (1,0), (1,1)],
...                      rabi_frequency=10, detuning=det, label="red")
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 4 states and 3 coherent␣
↪couplings.
States: [(0, 0), (1, -1), (1, 0), (1, 1)]
Coherent Couplings:
    ((0, 0),(1, -1)): {rabi_frequency: 10, detuning: <parameter with 11␣
```

(continues on next page)

(continued from previous page)

```
↪values>, phase: 0, kvec: (0, 0, 0), label: red_0, coherent_cc: 1.0, red_
↪detuning: detuning}
    ((0, 0),(1, 0)): {rabi_frequency: 10, detuning: <parameter with 11␣
↪values>, phase: 0, kvec: (0, 0, 0), label: red_1, coherent_cc: 1.0, red_
↪detuning: detuning}
    ((0, 0),(1, 1)): {rabi_frequency: 10, detuning: <parameter with 11␣
↪values>, phase: 0, kvec: (0, 0, 0), label: red_2, coherent_cc: 1.0, red_
↪detuning: detuning}
Decoherent Couplings:
    None
Energy Shifts:
    None
Zip Labels:
    ['red_detuning']
>>> print(s.get_hamiltonian().shape)
(11, 4, 4)
```

add_couplings(**couplings: Dict*, ***extra_kwargs*) → None

Add any number of couplings between pairs of states.

Acts as an alternative to calling `add_coupling()` individually for each pair of states. Can be used interchangeably up to preference, and all of keyword `add_coupling()` are supported dictionary keys for dictionaries passed to this function.

Note that since this function wraps `add_coupling()`, the first element of `couplings` will be used to set `probe_tuple`.

Parameters

- **couplings** (*tuple of dicts*) – Any number of dictionaries, each specifying the parameters of a single field coupling 2 states. For more details on the keys of each dictionary see the arguments for `add_coupling()`. Equivalent to passing each dictionaries keys and values to `add_coupling()` individually.
- ****extra_kwargs** (*dict*) – Additional keyword-only arguments to pass to the relevant `add_coupling` method. The same arguments will be passed to each call of `add_coupling()`. Often used for warning suppression. Can also be used to define a common coupling parameter for each coupling.

Examples

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s)
<class 'rydiqule.sensor.Sensor'> object with 3 states and 2 coherent␣
↪couplings.
States: [0, 1, 2]
Coherent Couplings:
    (0,1): {rabi_frequency: 1, detuning: 2, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: (0,1)}
    (1,2): {rabi_frequency: 3, detuning: 4, phase: 0, kvec: (0, 0, 0),␣
↪coherent_cc: 1.0, label: (1,2)}
Decoherent Couplings:
    None
Energy Shifts:
    None
```

add_decoherence(*statespecs: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], gamma: float | List[float] | ndarray, **kwargs*)

Add a coupling between states or groups of states.

Wraps the *add_single_decoherence()* and *add_decoherence_group()* functions, and dispatches to the appropriate one depending on the formatting of the `states` argument. Additional keyword arguments will be passed unmodified to the relevant method. See documentation of those functions for details on keyword argument options.

Parameters

- **states** (*tuple of StateSpec*) – The states or state manifolds of the decoherent coupling. If both are integers or string patterns matching a single state in the `Sensor`, *add_single_decoherence()* is dispatched. If either argument is a specification matching multiple states, *add_decoherence_group()* is dispatched.
- **gamma** (*float or Sequence*) – The decoherence rate, in Mrad/s.
- ****kwargs** – Additional keyword arguments passed to the appropriate function. See documentation for *add_single_decoherence()* and *add_decoherence_group()* for more details on valid keyword arguments.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), 0.1, label="misc")
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.  0. ]
 [0.1 0.  0. ]]
```

To add multiple decoherence effects to the same term, use a different label for each.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), 0.1, label='foo')
>>> s.add_decoherence((2,0), 0.15, label='bar')
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.   0.   0.  ]
 [0.25 0.   0.  ]]
```

Just like coherent coupling parameters, decoherence values can be passed as list-like objects and scanned. This adjusts the hamiltonian shape for clear broadcasting.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

Upper and lower states can also be regex strings matched against states in the Sensor just like for coherent couplings.

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> s.add_coupling(states=('g','e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=('e1','e2'), detuning=1, rabi_frequency=1)
>>> gamma = np.linspace(0, 0.3, 3)
>>> cc = {('e1','g'):0.25, ('e2','g'):0.75}
>>> s.add_decoherence((['e1','e2'],'g'), gamma, label="test", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[[0.     0.     0.    ]
  [0.     0.     0.    ]
  [0.     0.     0.    ]]

 [[0.     0.     0.    ]
  [0.0375 0.     0.    ]
  [0.1125 0.     0.    ]]

 [[0.     0.     0.    ]
  [0.075  0.     0.    ]
  [0.225  0.     0.    ]]]
```

Also just like coherent couplings, decoherent coupling can be defined over manifolds using state specifications. The interface is identical.

```
>>> g = (0,[-1,1])
>>> e = (1,[-1,1])
>>> cc = {
...     ((1,-1),(0,-1)):1,
...     ((1,1),(0,-1)):2,
...     ((1,-1),(0,1)):1,
...     ((1,1),(0,1)):2,
... }
>>> s = rq.Sensor([g,e])
>>> s.add_coupling((g,e), detuning=1, rabi_frequency=1, label='foo')
>>> s.add_decoherence((e,g), 0.1, coupling_coefficients=cc, label='bar')
>>> print(s.decoherence_matrix())
[[0.  0.  0.  0. ]
 [0.  0.  0.  0. ]
 [0.1 0.1 0.  0. ]
 [0.2 0.2 0.  0. ]]
```

add_decoherence_group(*states1: List[int | str | Tuple[float, ...]], states2: List[int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, label: str, coupling_coefficients: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], float] | None = None*)

Adds a group of decoherences to the Sensor.

Given 2 lists of states, adds a single coupling across each combination of states between the first and second lists. Then, if gamma is a array-like of values, automatically performs `zip_parameters()` on all decoherences added as part of this function so they share an axis when `decoherence_matrix()` is called.

Scaling multiplicative factors for `gamma` must be applied per pair of states using `decoherent_cc`, a dictionary of coefficients determining coupling strengths. If a pair is not in `decoherent_cc`, it is assumed to have a coupling coefficient of zero, and will be omitted from the graph. If `decoherent_cc` is `None`, all couplings are assumed to have a relative strength of 1.

Parameters

- **states1** (*List of State*) – The list of states out of which population is decaying.

Each element of the list must be a state in this `Sensor`.

- **states2** (*List of State*) – The list of states into which population is decaying. Each element of the list must be a state in this `Sensor`.
- **label** (*str*) – Required string label denoting what the group of dephasings is called. Used to apply a label to the zip.
- **gamma** (*ScannableParameter*) – Base decoherence rate between the two groups of states, in units of Mrad/s. Multiplied by the corresponding values in the `decoherent_coupling` dictionary.
- **coupling_coefficients** (*dict, optional*) – Coefficients describing the relative coupling strengths for decoherences in the group. Treated as modifications to the "base" dephasing rate specified by the `gamma` argument. The gamma of individual decoherences will be the `gamma` argument multiplied by the corresponding value in this dictionary. If `None`, all couplings in the group are assumed to have a coefficient of 1.0 if specified, all unspecified coupling pairs are ignored.

Raises

- **ValueError** – If the either of the states strings provided cannot be parsed as a regex pattern
- **ValueError** – If `states1` and `states2` only have one state. Use `add_decoherence()` instead.

Examples

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> s.add_coupling(states=('g','e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=('e1','e2'), detuning=1, rabi_frequency=1)
>>> cc = {('e1','g'):0.25, ('e2','g'):0.75}
>>> s.add_decoherence_group(['e1','e2'],['g'], 0.1, "test", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[0.    0.    0.   ]
 [0.025 0.    0.   ]
 [0.075 0.    0.   ]]
```

Unlike `Sensor.add_decoherence()`, this function does not accept state specifications. Upper and lower states must be passed as lists. As this tends to be a little clunkier, `Sensor.add_decoherence()` is usually preferred.

```
>>> g = (0,[-1,1])
>>> e = (1, [-1,1])
>>> list_g = [(0, -1), (0, 1)]
>>> list_e = [(1, -1), (1, 1)]
>>> cc = {
...     ((1,1),(0,1)): 0.4,
...     ((1,1),(0,-1)): 0.1,
...     ((1,-1),(0,1)): 0.1,
...     ((1,-1),(0,-1)): 0.4
... }
>>> s = rq.Sensor([g,e])
>>> print(s.states)
[(0, -1), (0, 1), (1, -1), (1, 1)]
>>> s.add_decoherence_group(list_e, list_g, 0.1, "foo", coupling_
↪coefficients=cc)
>>> print(s.decoherence_matrix())
[[0.   0.   0.   0.  ]
```

(continues on next page)

(continued from previous page)

```
 [0.   0.   0.   0.  ]
 [0.04 0.01 0.   0.  ]
 [0.01 0.04 0.   0.  ]]
```

add_energy_shift(*statespec: int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], shift: float | List[float] | ndarray, **kwargs*)

Add an energy shift to a single state or a group of states.

`statespec` can be provided either as a single state in the Sensor or as a valid state specification matching a group of states. When `statespec` matches a single state, `add_single_energy_shift()` method will be dispatched. In the case of a multi-state specification, the `add_energy_shift_group()` method will be dispatched applying an individual shift to all states with labels matching the specification provided.

Note that an energy shifts are applied to the underlying graph as a self-edge connecting a node to its self, not as data on the node its self.

Additional arguments for either dispatched function are passed normally via `**kwargs`

Parameters

- **state_spec** (*StateSpec*) – Integer or string label matching a state in the `Sensor`, or state specification matching one or more states in the `Sensor`. The number of states this corresponds to will affect which internal function is dispatched.
- **shift** (*float or array-like*) – The energy shift to apply to the matching state or states in Mrad/s. Note that if it corresponds to multiple states, the `prefactors` argument of `add_energy_shift_group()` will be multiplied by this value for the corresponding state.

Raises

RydiquleError – If the state provided does not match any state in the `Sensor`.

Examples

The basic use of `add_energy_shift` is to add terms to the diagonal of the hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_energy_shift(1, 1)
>>> s.add_energy_shift(2, 2.5)
>>> print(s.couplings.edges(data=True))
[(1, 1, {'e_shift': 1, 'label': '1'}), (2, 2, {'e_shift': 2.5, 'label': '2
↪'})]
>>> print(s.get_hamiltonian())
[[0. +0.j 0. +0.j 0. +0.j]
 [0. +0.j 1. +0.j 0. +0.j]
 [0. +0.j 0. +0.j 2.5+0.j]]
```

`add_energy_shift` can be used with state specifications.

```
>>> s = rq.Sensor([(0,0), (1,[-1,0,1])])
>>> prefactors = {(1,i):i for i in [-1,0,1]}
>>> s.add_energy_shift((1, [-1,0,1]), 0.1, prefactors=prefactors)
>>> print(s.couplings.edges(data="e_shift"))
[((1, -1), (1, -1), -0.1), ((1, 0), (1, 0), 0.0), ((1, 1), (1, 1), 0.1)]
>>> print(s.get_hamiltonian())
[[ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j -0.1+0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0. +0.j]
 [ 0. +0.j  0. +0.j  0. +0.j  0.1+0.j]]
```

add_energy_shift_group(*states: List[int | str | Tuple[float, ...]], shift: float | List[float] | ndarray, prefactors: dict | None = None, zip_label: str | None = None*)

Add energy shifts to a group of states, optionally with a modifying prefactor for each.

Given a list of states, calls `add_single_energy_shift()` on each one with the provided energy shift. Shifts are modified by a multiplicative factor defined by the `prefactors` dictionary. The dictionary is keyed with states that are elements of `states` with entries corresponding to a factor multiplied by the base `shift` argument for each state. When energy shifts are array-like, the `e_shift` attribute corresponding to each self-edge will be zipped with `zip_parameters()`.

Parameters

- **states** (*list of states*) – List of states to include in the group.
- **shift** (*float or array-like*) – The base value of the energy shift to apply the states. Will be modified by entries of the `prefactors` dictionary.
- **prefactors** (dict or `None`, optional) – Dictionary of values by which to multiply the base `shift` parameter for each each state. Keys are elements of the `states` list, entries are the corresponding factor by which to multiply `shift` for that state. If `None`, all prefactors are set to 1. If not `None`, the prefactors for any non-specified values will be set to zero. Default is `None`.
- **zip_label** (str or `None`, optional) – Label passed to `zip_parameters()` when the shift is provided as an array-like when all states in the group are zipped together. Defaults to `None`.

Raises

RydiquleError – If the supplied energy shift is not a float and cannot be interpreted as a numpy

Examples

```
>>> s = rq.Sensor(['g','e1','e2'])
>>> factors = {'e1':1, 'e2':2}
>>> s.add_energy_shift_group(["e1","e2"], 0.1, prefactors=factors)
>>> print(s.couplings.edges(data='e_shift'))
[('e1', 'e1', 0.1), ('e2', 'e2', 0.2)]
>>> print(s.get_hamiltonian())
[[0. +0.j 0. +0.j 0. +0.j]
 [0. +0.j 0.1+0.j 0. +0.j]
 [0. +0.j 0. +0.j 0.2+0.j]]
```

add_energy_shifts(*shifts: dict*)

Wrapper for `Sensor.add_energy shift b()`.

Shifts are specified with the `shifts` dictionary, which is keyed with states and has values corresponding to the energy shift applied to the state in Mrad/s. Error handling and validation is done with the `add_energy_shift()` function.

Parameters

shifts (*dict*) – Dictionary keyed with states with values corresponding to the energy shift, in Mrad/s, of the corresponding state.

add_self_broadening(*state: int | str | Tuple[float, ...], gamma: float | List[float] | ndarray, label: str = 'self', decoherent_cc: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], float] | None = None*)

Specify self-broadening (such as collisional broadening) of a level.

Equivalent to calling `add_decoherence()` and specifying both states to be the same, with the "self" label. For more complicated systems, it may be useful to further specify the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Parameters

- **state** (*State*) – State or states to which the broadening will be added. Using a regular expression allows for specifying self broadening of a group of states. In this case, `mult-factor` is used to define relative amplitudes.
- **gamma** (*float or sequence*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. By default, decay will be stored on the graph edge as `"gamma_self"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`
- **mult-factor** (*dict*) – Dictionary mapping of the scaling factors to apply to the self broadening of each state in a group specified via regular expression.

Notes

Note

Just as with the `add_decoherence()` function, adding a decoherence value with a label that already exists will overwrite an existing decoherent transition with that label. The "self" label is applied to this function automatically to help avoid an unwanted overwrite.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_self_broadening(1, 0.1)
>>> print(s.couplings.edges(data=True))
[(1, 1, {'gamma_self': 0.1, 'label': '(1,1)'})]
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.1 0. ]
 [0.  0.  0. ]]
```

add_self_broadening_group(*states: List[int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, label: str = 'self', decoherent_cc: dict | None = None*)

Specify self-broadening (such as collisional broadening) of a group of states.

Equivalent to calling `add_decoherence_group()` and specifying both state groups to be the same, with the "self" label. For more complicated systems, it may be useful to use `label` to label the source of self-broadening as, for example, "collisional" for easier bookkeeping and to ensure no values are overwritten.

Note that this function applies decoherence terms to every combination of states in the group, not just from each state to its self.

Parameters

- **states** (*list of State*) – List of states to which the self-broadening is applied.
- **gamma** (*ScannableParameter*) – The broadening width to be added in Mrad/s.
- **label** (*str, optional*) – Optional label for the state. By default, decay will be stored on the graph edge as `"gamma_self"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`
- **decoherent_cc** (*dict, optional*) – Clebsch-Gordon-like coefficients for how gamma scales to different pairs of states within the group. Unspecified pairs are assumed to have coefficients of 0. Default value is None, which applies 0 to all coefficients.

add_single_coupling(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], rabi_frequency: float | List[float] | ndarray | None = None, detuning: float | List[float] | ndarray | None = None, transition_frequency: float | None = None, phase: float | List[float] | ndarray | None = None, kvec: Sequence[float] = (0, 0, 0), time_dependence: Callable[[float], complex] | None = None, label: str | None = None, coherent_cc: float | None = None, **extra_kwargs*) → None

Adds a single coupling of states to the system.

One or more of these parameters can be a list or array-like of values to represent a laser that can take on a set of discrete values during a field scan. Designed to be a user-facing wrapper for `_add_coupling()` with arguments for states and coupling parameters.

Note that unlike `add_coupling()`, this function does not set the `probe_tuple` attribute, so if used to add the first coupling, `probe_tuple` must be set manually.

Parameters

- **states** (`tuple of States`) – The pair of states of the sensor which the state couples. Must be a tuple of length 2, where each element is a string, integer, or tuple corresponding to a state in the Sensor as defined in the constructor. Tuple order indicates which state to has higher energy; the second state is always assumed to have higher energy.
- **rabi_frequency** (`float or complex, or list-like of float or complex`) – The rabi frequency of the field being added. Defined in units of Mrad/s. List-like values will invoke Rydiqule's stacking convention when relevant quantities are calculated.
- **detuning** (`float or list-like of floats, optional`) – The frequency difference between the transition frequency and the field frequency in units of Mrad/s. List-like values will invoke Rydiqule's stacking convention when relevant quantities are calculated. If specified, the coupling is treated with the rotating-wave approximation rather than in the lab frame, and `transition_frequency` is ignored if present. A positive number always indicates a blue detuning, and a negative number indicates a blue detuning.
- **transition_frequency** (`float, optional`) – The transition frequency between a particular pair of states. Must be a positive number. Only used directly in calculations if `detuning` is `None`, ignored otherwise. Note that on its own, it only defines the spacing between two energy levels and not the field its self. To define a field, the `time_dependence` argument must be specified, or else the off-diagonal terms to drive transitions will not be generated in the Hamiltonian matrix.
- **phase** (`float, optional`) – Static phase offset in the rotating frame. Cannot be used outside the rotating frame, ie when detuning is not defined. Default is undefined, which is interpreted as 0 for couplings in the rotating frame.
- **kvec** (`iterable, optional`) – A three-element iterable that defines the k-vector of a particular coupling field. It should have units of Mrad/m, such that $vP * k_vec$ gives the most probable doppler shift along each axis. Note that the `vP` class attribute must be defined to perform doppler-broadened solves. If equal to `(0,0,0)`, solvers will ignore doppler shifts on this field. Defaults to `(0,0,0)`.
- **time_dependence** (`scalar function, optional`) – A scalar function specifying a time-dependent field. The time dependence function is defined as a python function that returns a unit-less value as a function of time (in microseconds) that is multiplied by the `rabi_frequency` parameter to get a field strength scaled to units of Mrad/s.
- **coherent_cc** (`float, optional`) – Additional information regarding coupling strength for the `states` coupling. Does **not** modify the `rabi_frequency` before adding to the graph. Rather, when the Hamiltonians and physical observables in `Solution` are computed, first multiplies the `rabi_frequency` by this value. The `rabi_frequency` can be thought of as a "base" field power, while this is a modification

based on the coupling strength. If `None`, the `coherent_cc` added to the graph will be set to 1. Defaults to `None`.

- **label** (*str or None, optional*) – Name of the coupling. This does not change any calculations, but can be used to help track individual couplings, and will be reflected in the output of `axis_labels()`, and to specify zipping for `zip_couplings()`. If `None`, the label is generated as the value of `states` cast to a string with whitespace removed. Defaults to `None`.

Raises

- ***RydiquleError*** – If `states` cannot be interpreted as a tuple.
- ***RydiquleError*** – If `states` does not have a length of 2.
- ***RydiquleError*** – If the states specified in the `states` argument are not in the basis of the `Sensor`
- ***RydiquleError*** – If both `rabi_frequency` and `dipole_moment` are specified or if neither are specified.
- ***RydiquleError*** – If both detuning and transition_frequency are specified or if neither are specified.
- ***RydiquleError*** – If a coupling is added in the non-rotating frame (detuning=None) and no time dependence function is specified.
- ***RydiquleError*** – If `kvec` is not a three element sequence of floats.

Warns

- **RWAWarning** – Raised if large `transition_frequency` is passed, which can lead to very long time-dependent solves and is often not intended.
- **RydiquleWarning** – Raised if 'kvec' is likely incorrectly defined (field k-vector in Mrad/m). Either by incorrect units or as most probable velocity vector (rq v1 convention). Triggers if the implied wavelength is less than 210 nm or greater than 2095 nm.

Examples

```
>>> s = rq.Sensor(2)
>>> s.add_single_coupling((0,1), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

```
>>> s = rq.Sensor(['g','e'])
>>> s.add_single_coupling(('g','e'), detuning=1, rabi_frequency=2)
>>> print(s.get_hamiltonian())
[[ 0.+0.j  1.+0.j]
 [ 1.-0.j -1.+0.j]]
```

```
>>> s = rq.Sensor(2)
>>> s.add_single_coupling((0,1), detuning=np.linspace(-10, 10, 101), rabi_
↪frequency=2, label="laser")
>>> print(s.get_hamiltonian().shape)
(101, 2, 2)
```

The `coherent_cc` attribute does not modify the `rabi_frequency` that is stored on the graph, but rather in the computed hamiltonian

```
>>> s = rq.Sensor(2)
>>> s.add_single_coupling((0,1), detuning=1, rabi_frequency=2, coherent_
↪cc=1/3)
>>> print(s.couplings.edges.data("rabi_frequency")) #shows the rabi_
↪frequency on the graph is 2
[(0, 1, 2)]
>>> print(s.get_hamiltonian())
[[ 0.        +0.j  0.333333+0.j]
 [ 0.333333-0.j -1.        +0.j]]
```

```
>>> s = rq.Sensor(2)
>>> step = lambda t: 1 if t>=1 else 0
>>> s.add_single_coupling((0,1), transition_frequency=1000, rabi_
↪frequency=2, time_dependence=step)
>>> print(s.get_hamiltonian())
[[   0.+0.j    0.+0.j]
 [   0.+0.j 1000.+0.j]]
>>> print(s.get_time_hamiltonian_components()[0])
[array([[0.+0.j, 2.+0.j],
       [2.-0.j, 0.+0.j]])]
```

```
>>> s = rq.Sensor(2, vP=10)
>>> kp = 25*np.array([1,0,0])
>>> s.add_single_coupling((0,1), detuning=1, rabi_frequency=2, kvec=kp)
>>> s.get_hamiltonian()
array([[ 0.+0.j,  1.+0.j],
       [ 1.-0.j, -1.+0.j]])
```

add_single_decoherence(*states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], gamma: float | List[float] | ndarray, decoherent_cc: float = 1.0, label: str | None = None*)

Add decoherent coupling to the graph between two states.

If `gamma` is list-like, the array generated by *decoherence_matrix()* will contain decoherence matrices for every combination of decoherence values provided. This functionality mirrors hamiltonian generation when parameters of *add_coupling()* are list-like. Note that if `gamma` is 0 or an array of zeros, the associated edge key will be left off the graph.

Parameters

- **states** (*tuple of State*) – Length-2 tuple of integers corresponding to the two states. The first value is the number of state out of which population decays, and the second is the number of the state into which population decays.
- **gamma** (*float or sequence*) – The decay rate, in Mrad/s.
- **decoherent_cc** (*float*) – The value by which `gamma` is multiplied before it is added to the graph. Typically only used by *add_decoherence_group()*, but made transparent for scripting purposes. Defaults to 1.0
- **label** (*str or None, optional*) – Optional label for the decay. If `None`, decay will be stored on the graph edge as `"gamma"`. Otherwise, will cast as a string and decay will be stored on the graph edge as `"gamma_"+label`.

Notes

Adding a decoherence with a particular label (including `None`) will override an existing decoherent transition with that label.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), 0.1, label="misc")
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.  0.  0. ]
 [0.1 0.  0. ]]
```

To add multiple decoherence effects to the same term, provide a different label for each.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), 0.1, label='foo')
>>> s.add_single_decoherence((2,0), 0.15, label='bar')
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.   0.   0.  ]
 [0.25 0.   0.  ]]
```

Decoherence values can also be scanned. Here decoherence from states 2->0 is scanned between 0 and 0.5 for 11 values. We can also see how the Hamiltonian shape accounts for this to allow for clean broadcasting, indicating that the hamiltonian is identical across all decoherence values.

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5,11)
>>> s.add_coupling(states=(0,1), detuning=1, rabi_frequency=1)
>>> s.add_coupling(states=(1,2), detuning=1, rabi_frequency=1)
>>> s.add_single_decoherence((2,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

add_single_energy_shift(*state: int | str | Tuple[float, ...], shift: float | List[float] | ndarray, label=None*)

Add an energy shift to a state.

First performs validation that the provided `state` is actually a node in the graph, then adds the shift specified by `shift` to a self-loop edge keyed with `"e_shift"`. This value will be added to the corresponding diagonal term when the hamiltonian is generated.

Parameters

- **state** (*int, str, or tuple*) – The label corresponding to the atomic state to which the shift will be added.
- **shift** (*float or list-like of float*) – The magnitude of the energy shift, in Mrad/s

Raises

RydiquleError – If the supplied `state` is not in the system.

add_transit_broadening(*gamma_transit: float | List[float] | ndarray*, *repop: Dict[int | str | Tuple[float, ...], float] | List[int | str | Tuple[float, ...]] | None = None*, *label: str = 'transit'*)

Adds transit broadening by adding a decoherence from each node to ground.

For each state n, adds a decoherent transition from n to each state in the keys of the `repop` dictionary using the `add_decoherence()` method with provided label (`"transit"`by default) See :meth:`~. Sensor.add_decoherence` for more details on labeling.

If an array of transit values are provided, they will be automatically zipped together into a single scanning element.

Parameters

- **gamma_transit** (*float or sequence*) – The transit broadening rate in Mrad/s.
- **repop** (*dict, optional*) – Dictionary of states for transit to repopulate in to. The keys represent the state labels. The values represent the fractional amount that goes to that state. If the sum of value does not equal 1, population will not be conserved. Default is to repopulate everything into the ground state (either state 0 or the first state in the basis passed to the `__init__()` method). If `None`, all population decays to the ground state, defined as the first state in the state list passed to the constructor. Defaults to `None`.
- **label** (*str, optional*) – Label to be passed to `add_decoherence()`. Defaults to "transit"

Warns

PopulationNotConservedWarning – If the values of the `repop` parameter do not sum to 1, thus meaning population will not be conserved.

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_transit_broadening(0.1)
>>> print(s.couplings.edges(data=True))
[(0, 0, {'gamma_transit': 0.1, 'label': '(0,0)'}),
 (1, 0, {'gamma_transit': 0.1, 'label': '(1,0)'}),
 (2, 0, {'gamma_transit': 0.1, 'label': '(2,0)'})]
>>> print(s.decoherence_matrix())
[[0.1 0.  0. ]
 [0.1 0.  0. ]
 [0.1 0.  0. ]]
```

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> repop = {'g':0.75, 'e1': 0.25}
>>> s.add_transit_broadening(0.2, repop=repop)
>>> print(s.decoherence_matrix())
[[0.15 0.05 0.  ]
 [0.15 0.05 0.  ]
 [0.15 0.05 0.  ]]
```

atom_mass: float | None = None

Mass of an atom in the vapor cell, in kilograms.

axis_labels() → List[str]

Get a list of axis labels for stacked hamiltonians.

The axes of a hamiltonian stack are defined as the axes preceding the usual hamiltonian, which are always the last 2. These axes only exist if one of the parameters used to define a Hamiltonian are lists.

Be default, labels which have been zipped using `zip_parameters()` will be combined into a single label, as this is how `get_hamiltonian()` treats these axes.

The ordering of axis labels is as follows:

- Zipped parameter (shared axes) appear before single parameters.
- Zipped parameters are ordered alphabetically by label.
- Single axes are sorted first by lower state, then by upper state, then alphabetically by parameter.

Returns
Strings corresponding to the label of each axis on a stack of multiple hamiltonians.

Return type
list of str

Examples

There are no preceding axes if there are no list-like parameters.

```
>>> s = rq.Sensor(3)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":2}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":4}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(3, 3)
>>> print(s.axis_labels())
[]
```

Adding list-like parameters expands the hamiltonian

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det, "label":
↪"blue"}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
>>> print(s.axis_labels())
['blue_detuning', '(1,2)_detuning']
```

The ordering of labels doesn't change if string state names are used. For single couplings, the ordering of axes is determined purely by the ordering of the states, regardless of coupling labels or string names of states.

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> det = np.linspace(-10, 10, 11)
>>> blue = {"states":('g','e1'), "rabi_frequency":1, "detuning":det, "label
↪":"blue"}
>>> red = {"states":('e1','e2'), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(blue, red)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
>>> print(s.axis_labels())
['blue_detuning', '(e1,e2)_detuning']
```

Zipping parameters combines labels onto a single axis, since their Hamiltonians now lie on a single axis of the stack. The name of that axis will be the label provided to `zipped_parameters()`. Note that this will default to 'zip_<int>'. Here the axis of length 7 (axis 1) corresponds to the rabi frequencies and the axis of shape 11 (axis 0) corresponds to the zipped detunings

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=np.
```

(continues on next page)

(continued from previous page)

```
↪linspace(-3, 3, 7))
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
>>> s.zip_parameters({(0,1):"detuning", (1,2):"detuning"}, zip_label=
↪"detunings")
>>> print(s.get_hamiltonian().shape)
(11, 7, 3, 3)
>>> print(s.axis_labels())
['detunings', '(0,1)_rabi_frequency']
```

property basis_size: int

Property to return the number of nodes on the Sensor graph.

Returns
The number of nodes on the graph, corresponding to the basis size for the system.

Return type
int

beam_area: float | None = None

Cross-sectional area of the probing beam, in square meters.

cell_length: float | None = None

Optical path length of the medium, in meters.

coupling_subgraph(*coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]]*) → Graph

Returns a subgraph view of the couplings graph corresponding to `coupling`.

Parameters
coupling (*StateSpecs*) – Coupling specification

Returns
View of the corresponding subgraph

Return type
networkx.Graph

couplings_with(**keys: str, method: Literal['all', 'any', 'not any'] = 'all'*) → Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Dict]

Returns a version of self.couplings with only the keys specified.

Can be specified with a several criteria, including all, none, or any of the keys specified.

Parameters

- **str)** (*keys (tuple of*) – parameter names for a state. See *add_coupling()* for which names are valid for a Sensor object.
- **method** (*{'all','any', 'not any'}*) – Method to see if a given field matches the keys given. Choosing "all" will return couplings which have keys matching all of the values provided in the keys argument, while choosing "any", will return all couplings with keys matching at least one of the values specified by keys. For example, `sensor.couplings_with("rabi_frequency")` returns a dictionary of all couplings for which a rabi_frequency was specified. `sensor.couplings_with("rabi_frequency", "detuning", method="all")` returns all couplings for which both rabi_frequency and detuning are specified. 'sensor.couplings_with("rabi_frequency", "detuning", method="any")` returns all couplings for which either rabi_frequency or detuning are specified. Defaults to "all".

Returns
A copy of the `sensor.couplings` dictionary with only couplings containing the specified parameter keys.

Return type
dict

Examples

Can be used, for example, to return couplings in the rotating wave approximation.

```
>>> s = rq.Sensor(3)
>>> sinusoid = lambda t: 0 if t<1 else sin(100*t)
>>> f2 = {"states": (0,1), "detuning": 1, "rabi_frequency":2}
>>> f1 = {"states": (1,2), "transition_frequency":100, "rabi_frequency":1,
↪"time_dependence": sinusoid}
>>> s.add_couplings(f1, f2)
>>> gamma = np.array([[.2,0,0],
...                   [.1,0,0],
...                   [0.05,0,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.couplings_with("detuning"))
{(0, 1): {'rabi_frequency': 2, 'detuning': 1, 'phase': 0, 'kvec': (0, 0,␣
↪0), 'coherent_cc': 1.0, 'label': '(0,1)'}}
```

decoherence_matrix() → ndarray

Build a decoherence matrix out of the decoherence terms of the graph.

For each edge, sums all parameters with a key that begins with "gamma", and places it on the appropriate location in an adjacency matrix for the `couplings` graph.

Returns
The decoherence matrix stack of the system.

Return type
numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> s.add_decoherence((1,0), 0.2, label="foo")
>>> s.add_decoherence((1,0), 0.1, label="bar")
>>> s.add_decoherence((2,0), 0.05)
>>> s.add_decoherence((2,1), 0.05)
>>> print(s.couplings.edges(data=True))
[(1, 0, {'gamma_foo': 0.2, 'label': '(1,0)', 'gamma_bar': 0.1}), (2, 0, {
↪'gamma': 0.05, 'label': '(2,0)'}), (2, 1, {'gamma': 0.05, 'label': '(2,1)
↪'})]
>>> print(s.decoherence_matrix())
[[0.   0.   0.  ]
 [0.3  0.   0.  ]
 [0.05 0.05 0.  ]]
```

Decoherences can be stacked just like any parameters of the Hamiltonian:

```
>>> s = rq.Sensor(3)
>>> gamma = np.linspace(0,0.5, 11)
>>> s.add_decoherence((1,0), gamma)
>>> print(s.decoherence_matrix().shape)
(11, 3, 3)
```

Defining decoherences between states labelled with string values works just like coherent couplings:

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> s.add_decoherence(('e1', 'g'), 0.1)
>>> s.add_decoherence(('e2', 'g'),0.1)
>>> print(s.decoherence_matrix())
[[0.  0.  0. ]
 [0.1 0.  0. ]
 [0.1 0.  0. ]]
```

dm_basis() → ndarray

Generate basis labels of density matrix components.

The basis corresponds to the elements in the solution. This is not the complex basis of the sensor class, but rather the real basis of a solution after calling one of `rydiqule`'s solvers. This means that the ground state population has been removed and it has been transformed to the real basis.

Returns
Array of string labels corresponding to the solving basis. Is a 1-D array of length `n**2-1`.

Return type
numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> print(s.dm_basis())
['10_real' '20_real' '10_imag' '11_real' '21_real' '20_imag' '21_imag'
 '22_real']
```

eta: float | None = None

Noise density prefactor, in units of root(Hz). Must be specified when using *Sensor*. Automatically calculated when using `Cell`.

get_couplings() → Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Dict]

Returns the couplings of the system as a dictionary

Deprecating in favor of calling the couplings.edges attribute directly.

Returns
A dictionary of key-value pairs with the keys corresponding to levels of transition, and the values being dictionaries of coupling attributes.

Return type
dict

get_doppler_shifts() → ndarray

Returns the Hamiltonian with only detunings set to the most probable doppler shift values for each spatial dimension.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns
Array of shape (used_spatial_dim,n,n), Hamiltonians with only the doppler shifts present along each non-zero spatial dimension specified by the fields' "kvec" parameter.

Return type
numpy.ndarray

get_hamiltonian() → ndarray

Creates the Hamiltonians from the couplings defined by the fields.

They will only be the steady state hamiltonians, i.e. will only contain terms which do not vary with time. Implicitly creates hamiltonians in "stacks" by creating a grid of all supported coupling parameters which are lists. This grid of parameters will not contain rabi-frequency parameters which vary with time and are defined as list-like. Rather, the associated axis will be of length 1, with the scanning over this value handled by the `get_time_couplings()` function.

For m list-like parameters x1,x2,…,xm with shapes N1,N2,…,Nm, and basis size n, the output will be shape `(N1,N2,...,Nm, n, n)`. The dimensions N1,N2,…Nm are labeled by the output of *axis_labels()*.

If any parameters have been zipped with the `_zip_parameters()` method, those parameters will share an axis in the final hamiltonian stack. In this case, if axis N1 and N2 above are the same shape and zipped, the final Hamiltonian will be of shape `(N1,...,Nm, n, n)`.

In the case where the basis of the `Sensor` was explicitly defined with a list of states, the ordering of rows and columns in the hamiltonian corresponds to the ordering of states passed in the basis.

See rydiqule's conventions for matrix stacking for more details.

Returns
The complex hamiltonian stack for the sensor.

Return type
np.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> blue = {"states":(0,1), "rabi_frequency":1, "detuning":det}
>>> red = {"states":(1,2), "rabi_frequency":3, "detuning":det}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 11, 3, 3)
```

Time dependent couplings are handled separately. The axis that contains array-like parameters with time dependence is length 1 in the steady-state Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
```

Zipping parameters means they share an axis in the Hamiltonian.

```
>>> s = rq.Sensor(3)
>>> s.add_coupling(states=(0,1), detuning=np.arange(11), rabi_frequency=2)
>>> s.add_coupling(states=(1,2), detuning=0.5*np.arange(11), rabi_
↪frequency=1)
>>> s.zip_parameters({(0,1):"detuning", (1,2):"detuning"})
>>> H = s.get_hamiltonian()
>>> print(H.shape)
(11, 3, 3)
```

If the basis is provided as a list of string labels, the ordering of Hamiltonian rows and columns will correspond to the order of states provided.

```
>>> s = rq.Sensor(['g', 'e1', 'e2'])
>>> s.add_coupling(('g', 'e1'), detuning=1, rabi_frequency=1)
>>> s.add_coupling(('e1', 'e2'), detuning=1.5, rabi_frequency=1)
>>> print(s.get_hamiltonian())
[[ 0. +0.j  0.5+0.j  0. +0.j]
 [ 0.5-0.j -1. +0.j  0.5+0.j]
 [ 0. +0.j  0.5-0.j -2.5+0.j]]
```

get_hamiltonian_diagonal(*values: dict*, *no_stack: bool = False*) → ndarray

Apply addition and subtraction logic corresponding to the direction of the couplings.

For a given state `n`, the path from ground will be traced to `n`. For each edge along this path, values will be added where the path direction and coupling direction match, and subtracting values where they do not. The sum of all such values along the path is the `n` th term in the output array.

Designed for internal functions which help generate hamiltonians. Most commonly used to calculate total detunings for ranges of couplings under the RWA

Parameters

- **values** (*dict*) – Key-value pairs where the keys correspond to transitions (agnostic to ordering of states) and values corresponding to the values to which the logic will be applied.
- **no_stack** (*bool, optional*) – Whether to ignore variable parameters in the system and use only basic math operations rather than reshape the output. Typically only `True` for calculating doppler shifts.

Returns

The diagonal of the hamiltonian of the system of shape `(*l,n)`, where `l` is the shape of the hamiltonian stack for the sensor.

Return type

numpy.ndarray

get_parameter_mesh() → List[ndarray]

Returns the parameter mesh of the sensor.

The parameter mesh is the flattened grid of variable parameters in all the couplings of a sensor. Wraps `numpy.meshgrid` with the `indexing` argument always `"ij"` for matrix indexing.

Returns

list of mesh grids for every variable parameter

Return type

list of numpy.ndarray

Examples

```
>>> s = rq.Sensor(3)
>>> rabi1 = np.linspace(-1,1,11)
>>> rabi2 = np.linspace(-2,2,21)
>>> s.add_coupling(states=(0,1), rabi_frequency=rabi1, detuning=1)
>>> s.add_coupling(states=(1,2), rabi_frequency=rabi2, detuning=1)
>>> for p in s.get_parameter_mesh():
...     print(p.shape)
(11, 1)
(1, 21)
```

get_rotating_frames() → dict

Determines the rotating frames for the disconnected subgraphs.

Each returned path gives the states traversed, and the sign gives the direction of the coupling. If the sign is negative, the coupling is going to a lower energy state. Choice of frame depends on graph distance to lowest indexed node on subgraph, ties broken by lowest indexed path traversed first.

Returns
Dictionary keyed by disconnected subgraphs, values are path dictionaries for each node of the subgraph. Each path shows the node indexes traversed, where a negative sign denotes a transition to a lower energy state.

Return type
dict

get_time_dependence() → List[Callable[[float], complex]]

Function which returns a list of the `time_dependence` functions.

The list is returned with in the order that matches with the time hamiltonians from `get_time_couplings()` such that the ith element of of the return of this functions corresponds with the ith Hamiltonian terms returned by that function.

Returns
List of scalar functions, representing all couplings specified with a `time_dependence`.

Return type
list

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> print(s.get_time_dependence())
[<function <lambda> at ...>, <function <lambda> at ...>]
```

get_time_hamiltonian(*t: float*) → ndarray

Get the system hamiltonian at a specific time, t.

This sums the steady-state hamiltonians with the time-dependent parts, evaluated at a specific time, t. If there is no time dependence in the system, function is equivalent to *get_hamiltonian()*.

Parameters
t (*float*) – Time to evaluate the time-dependence function at when building the hamiltonians

Returns
System hamiltonian, evaluated at time t.

Return type
numpy.ndarray

get_time_hamiltonian_components() → Tuple[List[ndarray], List[ndarray]]

Get time-dependent components of the hamiltonian.

Returns the list of matrices of all couplings in the system defined with a `time_dependence` key. The output will be two lists of matrices representing terms of the hamiltonian which are dependent on each time-dependent coupling. The lists will be of length M and shape `(*l_time, n, n)`, where M is the number of time-dependent couplings, `l_time` is time-dependent stack shape (possibly all ones), and `n` is the basis size. Each matrix will have terms equal to the rabi frequency (or half the rabi frequency under RWA) in positions that correspond to the associated transition. For example, in the case where there is a

`time_dependence` function defined for the `(2,3)` transition with a rabi frequency of 1, the associated time coupling matrix will be all zeros, with a 1 in the `(2,3)` and `(3,2)` positions.

Typically, this function is called internally and multiplied by the output of the `get_time_dependence()` function.

Returns

- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the real-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.
- *list of numpy.ndarray* – The list of M `(*l,n,n)` matrices representing the imaginary-valued time-dependent portion of the hamiltonian. For `0 <= i <= M`, the ith value along the first axis is the portion of the matrix which will be multiplied by the output of the ith `time_dependence` function.

Examples

```
>>> s = rq.Sensor(3)
>>> step = lambda t: 0 if t<1 else 1
>>> wave = lambda t: np.sin(2000*np.pi*t)
>>> f1 = {"states": (0,1), "transition_frequency":10, "rabi_frequency": 1,
↪"time_dependence":wave}
>>> f2 = {"states": (1,2), "transition_frequency":10, "rabi_frequency": 2,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2)
>>> time_hams, time_hams_i = s.get_time_hamiltonian_components()
>>> for H in time_hams:
...     print(H)
[[0.+0.j 1.+0.j 0.+0.j]
 [1.-0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 0.+0.j]]
[[0.+0.j 0.+0.j 0.+0.j]
 [0.+0.j 0.+0.j 2.+0.j]
 [0.+0.j 2.-0.j 0.+0.j]]
```

To handle stacking across the steady-state and time hamiltonians, the dimensions are matched in a way that broadcasting works in a numpy-friendly way

```
>>> s = rq.Sensor(3)
>>> rabi = np.linspace(-1,1,11)
>>> step = lambda t: 0 if t<1 else 1
>>> blue = {"states":(0,1), "rabi_frequency":rabi, "detuning":1}
>>> red = {"states":(1,2), "rabi_frequency":rabi, "detuning":0, 'time_
↪dependence': step}
>>> s.add_couplings(red, blue)
>>> time_hams, time_hams_i = s.get_time_hamiltonian_components()
>>> print(s.get_hamiltonian().shape)
(11, 1, 3, 3)
>>> print(time_hams[0].shape)
(1, 11, 3, 3)
>>> print(time_hams_i[0].shape)
(1, 11, 3, 3)
```

get_transition_frequencies() → ndarray

Gets an array of the diagonal elements of the Hamiltonian from the field detunings.

Wraps the `get_hamiltonian_diagonal()` function using both transition frequencies and detunings. Primarily for internal use.

Returns
N-D array of the hamiltonian diagonal. For an n-level system with stack shape `*l`, will be shape `(*l, n)`

Return type
numpy.ndarray

get_value_dictionary(*key: str*) → dict

Get subset of dictionary coupling parameters.

Return a dictionary of key value pairs where the keys are couplings added to the system and the values are the value of the parameter specified by key. Produces an output that can be passed directly to `get_hamiltonian_diagonal()`. Only couplings whose parameter dictionaries contain "key" will be in the returned dictionary.

Parameters
key (*str*) – String value of the parameter name to build the dictionary. For example, `get_value_dictionary("detuning")` will return a dictionary with keys corresponding to transitions and values corresponding to detuning for each transition which has a detuning.

Returns
Coupling dictionary with couplings as keys and corresponding values set by input key.

Return type
dict

Examples

```
>>> s = rq.Sensor(4)
>>> f1 = {"states": (0,1), "detuning": 2, "rabi_frequency": 1}
>>> f2 = {"states": (1,2), "detuning": 3, "rabi_frequency": 2}
>>> step = lambda t: 1 if t>1 else 0
>>> f3 = {"states": (2,3), "rabi_frequency": 3, "transition_frequency": 3,
↪"time_dependence":step}
>>> s.add_couplings(f1, f2, f3)
>>> print(s.get_value_dictionary("detuning"))
{(0, 1): 2, (1, 2): 3}
```

group_variable_parameters(*apply_mesh: bool = False*) → List[List[Tuple[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], str, ndarray, str | None]]]

int_states_map(*invert: bool = False*) → Dict[int | str | Tuple[float, ...], int] | Dict[int, int | str | Tuple[float, ...]]

Get a dictionary mapping between state labels and their corresponding integer ordering. Can be returned with `key:value` pairs defined either by `label:int` or `int:label`, controlled via optional `invert` argument.

Parameters
invert (*bool, optional*) – Whether to switch the role of keys and values. Labels are keys if `False`, and values if `True`, by default False

Returns
Dictionary mapping between state labels and integer ordering

Return type
dict

kappa: float | None = None

Differential prefactor, in units of (rad/s)/m. Must be specified when using `Sensor`. Automatically calculated when using `Cell`.

probe_freq: float | None = None

Probing transition frequency, in rad/s.

property probe_tuple: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]] | None

Coupling edge that corresponds to the probing field. Defaults to `None` and gets set to the first coupling added to the system with `add_coupling()`. Can be modified directly.

set_experiment_values(*probe_freq: float, kappa: float, eta: float | None = None, cell_length: float | None = None, beam_area: float | None = None, v_th: float | None = None, temp: float | None = None, atom_mass: float | None = None*)

Sets attributes needed for observable calculations.

Parameters

- **probe_tuple** (*tuple of StateSpec*) – Coupling that corresponds to the probing field. If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`.
- **probe_freq** (*float*) – Frequency of the probing transition, in rad/s. If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`.
- **kappa** (*float*) – Numerical prefactor that defines susceptibility, in (rad/s)/m. If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`. See `get_susceptibility()` and `Cell.kappa` for details.
- **eta** (*float*) – Noise-density prefactor, in root(Hz). If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`. See `Cell.eta` for details.
- **cell_length** (*float, optional*) – The optical path length through the medium, in meters. If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`.
- **beam_area** (*float, optional*) – The cross-sectional area of the beam, in m^2. If `None`, corresponding Sensor attribute remains unchanged. Defaults to `None`.
- **v_th** (*float, optional*) – Thermal velocity of the Maxwell-Boltzmann distribution: v_th = (k_B*T/m)^(1/2) in units of m/s. Defaults to `None`.
- **temp** (*float, optional*) – Temperature of the vapor cell in units of Kelvin. Defaults to `None`.
- **atom_mass** (*float, optional.*) – Mass of a sensing atom in kg. Defaults to `None`.

set_gamma_matrix(*gamma_matrix: ndarray*)

Set the decoherence matrix for the system.

Works by first removing all existing decoherent data from graph edges, then individually adding all nonzero terms of a provided gamma matrix to the corresponding graph edges. Can be used to set all decoherence attributes to edges simultaneously, but `add_decoherence()` is preferred.

Unlike `add_decoherence()`, does not support scanning multiple decoherence values, rather should be used to set the decoherences of the system to individual static values.

Parameters

gamma_matrix (*numpy.ndarray*) – Array of shape `(basis_size, basis_size)`. Element `(i,j)` describes the decoherence rate, in Mrad/s, from state `i` to state `j`.

Raises

- **RydiquleError** – If `gamma_matrix` is not a numpy array.
- **ValueError** – If `gamma_matrix` is not a square matrix of the appropriate size
- **ValueError** – If the shape of `gamma_matrix` is not compatible with `self.basis_size`.

Examples

```
>>> s = rq.Sensor(2)
>>> f1 = {"states": (0,1), "detuning":1, "rabi_frequency": 1}
>>> s.add_couplings(f1)
>>> gamma = np.array([[.1,0],[.1,0]])
>>> s.set_gamma_matrix(gamma)
>>> print(s.decoherence_matrix())
[[0.1 0. ]
 [0.1 0. ]]
```

spatial_dim() → int

Returns the number of spatial dimensions doppler averaging will occur over.

Determining if a float should be treated as zero is done using `numpy.isclose`, which has default absolute tolerance of `1e-08`.

Returns
Number of dimensions, between 0 and 3, where 0 means no doppler averaging k-vectors have been specified or are too small to be calculates.

Return type
int

Examples

No spatial dimensions specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1)
>>> print(s.spatial_dim())
0
```

One spatial dimension specified

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(0,0,4))
>>> print(s.spatial_dim())
1
```

Multiple spatial dimensions can exist in a single coupling or across multiple couplings

```
>>> s = rq.Sensor(2)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(3,0,3))
>>> print(s.spatial_dim())
2
```

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1, kvec=(3,0,3))
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, kvec=(0,4,0))
>>> print(s.spatial_dim())
3
```

property states: List[int | str | Tuple[float, ...]]

Property which gets a list of labels for the sensor in the order defined in `__init__()`. This is also the order corresponding the rows and columns in the system Hamiltonian and decoherence matrix.

Returns
List of states of the system defined the constructor, in the order corresponding to rows and columns of the Hamiltonian.

Return type
list

states_with_spec(*statespec: int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]*) → List[int | str | Tuple[float, ...]]

Return a list of all states in the sensor matching the `state_spec` pattern.

A state is considered a "match" if, for each element of the state, the corresponding element of `statespec` is either exactly the floating point or string value, or a list containing that element of state. In this way, groups of states can be specified more tersely than a complete list of all states.

Parameters
statespec – The StateSpec against which state labels in sensor are to be matched

Returns
All the states in the sensor matching the given specification.

Return type
list of State

Examples

```
>>> states = [
...     (0,0),
...     (1,-1),
...     (1,0),
...     (1,1)
... ]
>>> s = rq.Sensor(states)
>>> s.states_with_spec((1,[-1,0,1]))
[(1, -1), (1, 0), (1, 1)]
```

temp: float | None = None

Temperature of the vapor cell, in Kelvin.

unzip_parameters(*zip_label: str, verbose: bool | None = True*)

Remove a set of zipped parameters from the internal zip_labels list.

If an element of the internal `_zip_labels` array matches the label provided, removes it from `_zip_labels`. If no such element is present in `_zip_labels`, does nothing, and prints a message (disabled with `verbose=False`)

Parameters

- **zip_label** (*str*) – The string label corresponding the key to be deleted in the `_zip_labels` attribute.
- **verbose** (*bool*) – Whether to print a message if the unzip fails due to the specified `zip_label` not being a zip in the sensor. If `True` prints a message to std out if `zip_label` is not an element of the internal `self._zip_labels`. Otherwise, fails silently. Can be used if unzipping as part of an automated script.

Notes

Note

This function should always be used rather than modifying the `_zip_labels` attribute directly.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"}, zip_label=
↪"demo1")
>>> print(s._zip_labels) #NOT modifying directly
['demo1']
>>> print(s.couplings.edges(data="demo1"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> s.unzip_parameters("demo1")
>>> print(s._zip_labels) #NOT modifying directly
[]
>>> print(s.couplings.edges(data="demo1"))
[(0, 1, None), (1, 2, None)]
```

If the labels provided are not a match, a message is printed and nothing is altered. In the case where simulations are scripted and the printed message is annoying, the print behavior can be modified with `verbose=False`, potentially useful for scripting cases where the desired behavior is to silently continue over non-existent zip labels.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"})
>>> print(s._zip_labels) #NOT modifying directly
['zip_0']
>>> print(s.couplings.edges(data="zip_0"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> s.unzip_parameters("zipp0")
No label matching zipp0, no action taken
>>> print(s._zip_labels) #NOT modifying directly
['zip_0']
>>> print(s.couplings.edges(data="zip_0"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
```

property vP: float

Most probable speed of the 3D Maxwell-Boltzmann distribution.

This is defined as $\sqrt{2kT/m}$ and is given in units of m/s.

This must be defined manually when performing Doppler-broadened solves. Accessing it before definition will raise an error.

v_th: float | None = None

Thermal velocity of the atoms in vapor cell, in meters per second.

variable_parameter_sort(*par: tuple*) → tuple

Assistance function which determines the sorting order of elements parameters in sensor.

Called in *variable_parameters()* to ensure a consistent sort order. Provided as the `key` parameter in python's `sorted()` function before parameters are returned.

Sorts first by `zip_label`, then by `states`, then by `parameter`. Ensures all parameters zipped with one another are grouped together in a list. Zipped parameters will always come first. From there, parameters

are sorted alphabetically by zip_label (including case), then by state pair (as determined by ordering in the sensor, NOT alphabetically), then alphabetically by parameter.

Parameters
par (*tuple*) – 4-element list of information on each parameter. Consists of `(states, parameter, value, zip_label)`

Returns
3-element tuple that defines a particular parameter's position in the final sorting order.

Return type
tuple

variable_parameters(*apply_mesh: bool = False*) → List[Tuple[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], str, ndarray, str | None]]

Property to retrieve the values of parameters that were stored on the graph as arrays.

Values are returned as a list of tuples in the standard order of pythons default sorting, applied first to the tuple indicating states and then to the key of the parameter itself. This means that couplings are sorted first by lower state, then by upper state, then alphabetically by the name of the parameter.To determine order, all state labels treated as their integer position in the basis as determined by ordering in the constructor `__init__()`.

Returns
A list of tuples corresponding to the parameters of the systems that are variable (i.e. stored as an array). They are ordered according to states, then according to variable name. Tuple entries of the list take the form `(states, param_name, value)`

Return type
list of tuples

Examples

```
>>> s = rq.Sensor(3)
>>> vals = np.linspace(-1,2,3)
>>> s.add_coupling(states=(1,2), rabi_frequency=vals, detuning=1)
>>> s.add_coupling(states=(0,1), rabi_frequency=vals, detuning=vals)
>>> print(s.variable_parameters())
[((0, 1), 'detuning', array([-1. ,  0.5,  2. ]), None),
 ((0, 1), 'rabi_frequency', array([-1. ,  0.5,  2. ]), None),
 ((1, 2), 'rabi_frequency', array([-1. ,  0.5,  2. ]), None)]
```

The order is important; in the unzipped case, it will sort as though all state labels were cast to strings, meaning integers will always be treated as first.

```
>>> s = rq.Sensor([0, 'e1', 'e2'])
>>> det1 = np.linspace(-1, 1, 3)
>>> det2 = np.linspace(-1, 1, 5)
>>> blue = {"states":(0,'e1'), "rabi_frequency":1, "detuning":det1}
>>> red = {"states":('e1','e2'), "rabi_frequency":3, "detuning":det2}
>>> s.add_couplings(blue, red)
>>> print(s.variable_parameters())
[((0, 'e1'), 'detuning', array([-1.,  0.,  1.]), None),
 (('e1', 'e2'), 'detuning', array([-1. , -0.5,  0. ,  0.5,  1. ]), None)]
>>> print(f"Axis Labels: {s.axis_labels()}")
Axis Labels: ['(0,e1)_detuning', '(e1,e2)_detuning']
```

zip_parameters(*parameters: Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]] | str, str], zip_label: str | None = None*)

Define 2 scannable parameters as "zipped" so they are scanned in parallel.

Zipped parameters will share an axis when quantities relevant to the equations of motion, such as the gamma_matrix and hamiltonian are generated. So for 2 list-like parameters, the first elements in each are solved at the same time, then the second, etc Note that calling this function does not affect internal quantities directly, but flags them to be zipped at calculation time for relevant quantities.

Internally, adds the label value to the internal list of zipped parameter labels, and adds a flag in the form of <label>:<parameter_name> to each edge of the graph.

Parameters

- **parameters** (*dict*) – Parameter labels to scan together. Parameters are specified with a dictionary keyed by the either pair of states defining the coupling (e.g. (0,1)) or a previously specified label (e.g. "probe") with items corresponding to the respective parameter name (e.g. "detuning").
- **zip_label** (*optional, str*) – String label shorthand for the zipped parameters. The label for the axis of these parameters in *axis_labels()*. Does not affect functionality of the Sensor. If None (the default), the label used will be "zip_" + <number>, where <number> is the one index beyond the current length of the zip_parameters list.

Raises

- ***RydiquleError*** – If fewer than 2 labels are provided.
- ***RydiquleError*** – If any of the 2 labels are the same.
- ***RydiquleError*** – If the label contains the substring "gamma", as this is used internally for decoherence matrix generation.
- ***RydiquleError*** – If any elements of labels are not labels of couplings in the sensor.
- ***RydiquleError*** – If any of the parameters specified by labels are already zipped.
- ***RydiquleError*** – If any of the parameters specified are not list-like.
- ***RydiquleError*** – If all list-like parameters are not the same length.

Notes

Note

This function should be called last after all Sensor couplings and dephasings have been added. Changing a coupling that has already been zipped removes it from the self.zipped_parameters list.

Note

Modifying the Sensor._zip_labels attribute directly can break some functionality and should be avoided. Use this function or *unzip_parameters()* instead.

Note

When defining the zip strings for states labelled with strings, be sure to additional ' or " characters on either side of the labels, as demonstrated in the second example below.

Examples

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling(states=(0,1), detuning=det, rabi_frequency=1, label=
```

(continues on next page)

(continued from previous page)

```
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=det, rabi_frequency=1)
>>> s.zip_parameters({"probe":"detuning", (1,2):"detuning"}, zip_label=
↪"detunings")
>>> print(s._zip_labels) #NOT modifying directly
['detunings']
>>> print(s.couplings.edges(data="detunings"))
[(0, 1, 'detuning'), (1, 2, 'detuning')]
>>> print(s.get_hamiltonian().shape)#zipped parameters share an axis
(11, 3, 3)
```

Especially when states are labelled with tuples, specifying zips parameters with the states they couple can be cumbersome. In this case, it can be useful to either assign variables to the tuples defining the states, or to label the couplings.

```
>>> g = (0,0)
>>> e1, e2 = (1,-1), (1, 1)
>>> s = rq.Sensor([g, e1, e2])
>>> arr = np.linspace(-1,1,11)
>>> s.add_coupling((g,e1), detuning=arr, rabi_frequency=1, label="probe")
>>> s.add_coupling((e1,e2), detuning=arr, rabi_frequency=1, label="coupling
↪")
>>> s.zip_parameters({((0,0),(1,-1)):"detuning", ((1,-1), (1, 1)):"detuning
↪"}, zip_label="foo") #clunky
>>> print(s._zip_labels)
['foo']
>>> s.unzip_parameters("foo")
>>> s.zip_parameters({"probe":"detuning", "coupling":"detuning"}, zip_
↪label="bar") #readable
>>> print(s._zip_labels)
['bar']
```

For maximum flexibility, any parameters specified as arrays with matching lengths can be zipped. This should be used with care, as some parameter combinations can be nonsensical.

```
>>> s = rq.Sensor(3)
>>> arr = np.linspace(-1,1,11)
>>> s.add_energy_shift(0, 0.5*arr)
>>> s.add_coupling(states=(0,1), detuning=arr, rabi_frequency=1, label=
↪"probe")
>>> s.add_coupling(states=(1,2), detuning=arr, rabi_frequency=1)
>>> s.add_decoherence((1,0), 0.1*arr)
>>> s.zip_parameters({(0,0):"e_shift", "probe":"detuning", (1,2):"detuning
↪", (1,0):"gamma"}, zip_label="foo")
>>> print(s._zip_labels) #NOT modifying directly
['foo']
>>> print(s.couplings.edges(data="foo"))
[(0, 0, 'e_shift'), (0, 1, 'detuning'), (1, 2, 'detuning'), (1, 0, 'gamma
↪')]
>>> print(s.get_hamiltonian().shape)
(11, 3, 3)
```

zip_zips(**zip_labels: str*, *new_label: str | None = None*)

Combine multiple parameter zips into a single zip.

Given any number of labels of zips in the sensor, combines them so that they will all share a single axis in the stack. Note that this will override all previous zips in `zip_labels`, and they cannot be recovered.

Parameters

new_label (*string, optional*) – Label for the new zip that will replace the ones provided in `zip_labels`. If None, will be generated by joining all the strings of `zip_labels` with a "_" character, by default None.

Raises

- ***RydiquleError*** – If any of `zip_labels` do not exist in the `Sensor`.
- ***RydiquleError*** – If `new_label` contains a protected substring (such as "gamma").
- ***RydiquleError*** – If any of `zip_labels` are the same.
- ***RydiquleError*** – If any of the dimensions of the axes specified by `zip_labels` do not match.

Examples

```
>>> s = rq.Sensor(5)
>>> det = np.linspace(-1, 1, 11)
>>> s.add_coupling((0, [1,2]), rabi_frequency=1, detuning=det, label="foo")
>>> s.add_coupling((0, [3,4]), rabi_frequency=1, detuning=det, label="bar")
>>> print(s.get_hamiltonian().shape)
(11, 11, 5, 5)
>>> print(s.axis_labels())
['bar_detuning', 'foo_detuning']
>>> s.zip_zips("foo_detuning", "bar_detuning", new_label="foobar_detuning")
>>> print(s.get_hamiltonian().shape)
(11, 5, 5)
>>> print(s.axis_labels())
['foobar_detuning']
```

8.1.10 rydiqule.sensor_solution

Object used to store aspects of a solution when calling rydiqule.solve() Adds essential keys with "None" entries

Classes

Solution()	Results object that contains information from a solve.

rydiqule.sensor_solution.Solution

class rydiqule.sensor_solution.**Solution**

Bases: `object`

Results object that contains information from a solve.

Methods implement a number of standard analyses on the result based on the density matrix observable formalism and Maxwell's equations for a plane wave in an optically-thin polarizable medium.

__init__()

Methods

__init__()	
copy()	
coupling_coefficient_matrix(coupling)	Matrix of coefficients representing coupling strength for a particular coupling.

continues on next page

Table 8.36 – continued from previous page

coupling_coefficient_observable([coupling])	Get the observable associated with the output of coupling_coefficient_matrix().
coupling_rabi(coupling)	Rabi frequency, in Mrad/s, of a particular coupling or coupling group.
deepcopy()	
get_OD()	Calculates the optical depth from the solution.
get_observable(matrix)	Returns the trace of a matrix product of the density matrix with a provided matrix.
get_phase_shift()	Extract the phase shift from a solution.
get_solution_element(idx)	Return a slice of an n_dimensional matrix of solutions of shape (...,n^2-1), where n is the basis size of the quantum system.
get_susceptibility()	Return the atomic susceptibility on the probe transition.
get_transmission_coef()	Extract the transmission term from a solution.
rho_ij(i, j)	Gets the full complex density matrix element corresponding to a given pair of indices.

Attributes

beam_area	Cross-sectional area of the probing beam, in square meters.
cell_length	Optical path length of the medium, in meters.
complex_rho	Density-matrix solutions in complex basis.
eta	Eta constant from the Cell.
kappa	Kappa constant from the Cell.
probe_freq	Probing transition frequency, in rad/s.
probe_rabi	Base laser rabi frequency of the probing transition of the Sensor used in a solve.
probe_tuple	Coupling edge corresponding to probing field.
states	Return a list of all states in the Sensor used to produce solution.
couplings	Copy of the Sensor.couplings graph.
axis_labels	Labels for the axes of scanned parameters.
axis_values	Value corresponding to each axis.
rq_version	Version of rydiqule that created the Solution.
dm_basis	The list of density matrix elements in the order they appear in the solution.
variable_parameters	Output of the Sensor.variable_parameters() method for the Sensor used in a solve.
doppler_classes	Doppler classes used to perform the doppler average.
rho	Solutions returned by the solver.
t	Times the solution is returned at, when using the time solver.
init_cond	Initial conditions, when using the time solver.

_beam_area: float | None

Cross-sectional area of the probing beam, in square meters. Not generally defined when using a Sensor.

_cell_length: float | None

Optical path length of the medium, in meters. Not generally defined when using a Sensor.

_eta: float | None

Eta constant from the Cell. Not generally defined when using a Sensor.

Type
float, optional

_kappa: float | None

Kappa constant from the Cell. Not generally defined when using a Sensor.

Type
float, optional

_probe_freq: float | None

Probing transition frequency, in rad/s. Not generally defined when using a Sensor.

_probe_tuple: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]] | None

Coupling edge corresponding to probing field. Defined as the first added coupling when using a Sensor.

axis_labels: list[str]

Labels for the axes of scanned parameters. If doppler averaging but not summing, doppler dimensions are prepended.

Type
list of str

axis_values: list

Value corresponding to each axis. If doppler averaging but not summing, doppler classes in internal units are added.

Type
list

property beam_area: float

Cross-sectional area of the probing beam, in square meters. Not generally defined when using a Sensor.

property cell_length: float

Optical path length of the medium, in meters. Not generally defined when using a Sensor.

property complex_rho: ndarray

Density-matrix solutions in complex basis.

This function calls *sensor_utils.convert_dm_to_complex()* on *rho*.

Returns
Complex density matrix solutions, with ground state present.

Return type
numpy.ndarray

copy()

coupling_coefficient_matrix(*coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]]*) → ndarray

Matrix of coefficients representing coupling strength for a particular coupling.

Returns
Adjacency matrix of `couplings` attributes generated from the `"coherent_cc"` parameter on the `coupling` edge.

Return type
np.ndarray

Examples

```
>>> g = rq.ground_state(5, splitting="fs")
>>> e = rq.D1_excited(5, splitting="fs")
>>> my_cell = rq.Cell('Rb85', [g, e])
>>> my_cell.add_coupling(states=(g, e), rabi_frequency=1, detuning=1,␣
↪label="probe")
>>> sol = rq.solve_steady_state(my_cell)
>>> for e in sol.couplings.edges(data="coherent_cc"):
...     print(*e)
(5, 0, 0.5, m_j=-0.5) (5, 0, 0.5, m_j=-0.5) None
(5, 0, 0.5, m_j=-0.5) (5, 0, 0.5, m_j=0.5) None
(5, 0, 0.5, m_j=-0.5) (5, 1, 0.5, m_j=-0.5) -0.816496580927726
(5, 0, 0.5, m_j=0.5) (5, 0, 0.5, m_j=-0.5) None
(5, 0, 0.5, m_j=0.5) (5, 0, 0.5, m_j=0.5) None
(5, 0, 0.5, m_j=0.5) (5, 1, 0.5, m_j=0.5) 0.816496580927726
(5, 1, 0.5, m_j=-0.5) (5, 0, 0.5, m_j=-0.5) None
(5, 1, 0.5, m_j=-0.5) (5, 0, 0.5, m_j=0.5) None
(5, 1, 0.5, m_j=0.5) (5, 0, 0.5, m_j=-0.5) None
(5, 1, 0.5, m_j=0.5) (5, 0, 0.5, m_j=0.5) None
>>> print(sol.coupling_coefficient_matrix(sol.probe_tuple))
[[ 0.         0.         0.         0.       ]
 [ 0.         0.         0.         0.       ]
 [-0.816497  0.         0.         0.       ]
 [ 0.         0.816497  0.         0.       ]]
```

coupling_coefficient_observable(*coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]] | None = None*)

Get the observable associated with the output of `coupling_coefficient_matrix()`.

Calls the `get_observable()` function with the output of `coupling_coefficient_matrix()`, with rows and columns limited those defined by the probe coupling. If `coupling` is `None`, uses the tuple defined by `probe_tuple`.

This function is designed to get observable values associated with the probe laser in an experiment, such as transmission and absorption coefficients.

Parameters

coupling (*tuple of int or string, optional*) – The 2-length tuple of state specifications to use in the observable calculation. Each state can be either an `int` or a regex string corresponding to a group of states. If `None`, uses the `probe_tuple` attribute as defined in the `Sensor` used to produce the solution. Defaults to `None`.

Returns

The observable or array of observables corresponding to the coupling coefficients.

Return type

np.ndarray

coupling_rabi(*coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]]*) → complex | ndarray

Rabi frequency, in Mrad/s, of a particular coupling or coupling group.

Serves as a more general way of fetching a rabi frequency from the graph that supports couplings between manifolds. Throws an error if any of the couplings in the group specified by `coupling` do not have matching `rabi_frequency`, either in values or shape.

Parameters

coupling (*tuple of states*) – Pair of states or state manifolds specifying a coupling or

coupling group respectively. If a group, all couplings in the group must have matching values and shapes for the `rabi_frequency`. This is most easily accomplished by adding the coupling to the group using `add_coupling()` with manifolds, but `zip_parameters()` can also be used.

Returns
The `rabi_frequency` of the coupling or of all couplings in the group specified by `coupling`.

Return type
float or np.ndarray

Raises
ValueError – If `coupling` is a group with mismatched rabi frequencies, either in shape or value, or coupling has no defined rabi frequencies.

Examples

The basic functionality for couplings between single states mimics the functionality of just accessing the `rabi_frequency` attribute from the graph.

```
>>> my_sensor = rq.Sensor(2)
>>> my_sensor.add_coupling((0,1), rabi_frequency=1, detuning=1)
>>> my_sensor.add_decoherence((1,0), 0.1)
>>> sol = rq.solve_steady_state(my_sensor)
>>> print(sol.couplings.edges[0,1]["rabi_frequency"])
1
>>> print(sol.coupling_rabi((0,1)))
1
```

It can also get the `rabi_frequency` for an entire coupling group. The `rabi_frequency` is treated as a laser property. Coupling coefficients as individual coupling properties, and so are not accounted for.

```
>>> g = (0,0)
>>> e = (1, [-1,1])
>>> cc = {
...     (g, (1,-1)): 0.5,
...     (g, (1,1)): 0.5
... }
>>> my_sensor = rq.Sensor([g,e])
>>> my_sensor.add_coupling((g,e), rabi_frequency=5, detuning=1, label=
↪"probe")
>>> my_sensor.add_decoherence((e,g), 0.1, label="foo")
>>> sol = rq.solve_steady_state(my_sensor)
>>> print(sol.couplings.edges.data("rabi_frequency"))
[((0, 0), (1, -1), 5), ((0, 0), (1, 1), 5), ((1, -1), (0, 0), None), ((1,␣
↪1), (0, 0), None)]
>>> print(sol.coupling_rabi((g,e)))
5
```

couplings: DiGraph

Copy of the `Sensor.couplings` graph.

Type
dict

deepcopy()

dm_basis: ndarray

The list of density matrix elements in the order they appear in the solution. See `Sensor.basis()` for details.

Type
list of str

doppler_classes: ndarray | None

Doppler classes used to perform the doppler average. Will be None if doppler averaging was not used.

Type
numpy.ndarray, optional

property eta: float

Eta constant from the Cell. Not generally defined when using a Sensor.

get_OD() → float | ndarray

Calculates the optical depth from the solution. This equation comes from Steck's Quantum Optics Notes Eq. 6.74.

Assumes the optically-thin approximation is valid. If a calculated OD for a solution exceeds 1, this approximation is likely invalid.

Returns
OD – Optical depth of the sample

Return type
float or numpy.ndarray

Warns
UserWarning – If any OD exceeds 1, which indicates the optically-thin approximation is likely invalid.

Examples

```
>>> [g, e] = rq.D2_states('Rb85')
>>> c = rq.Cell('Rb85', [g, e], cell_length =  0.0001)
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
(3,)
>>> OD = sols.get_OD()
>>> print(OD)
0.2964844
```

get_observable(*matrix: ndarray*)

Returns the trace of a matrix product of the density matrix with a provided matrix. This is the standard definition of an measurement of observable A taken on a density matrix ρ given by $tr(\rho A)$. Note that this function first converts the density matrix into the standard complex basis rather than rydiqule's real basis, leading to potential round-off errors for *very* small density matrix elements.

The provided observable wil respect `rydiqule` stacking convention, and the labels for each axis can be recovered via the `axis_labels` attribute.

Parameters
matrix (*np.ndarray*) – The matrix representing the observable to be computed.

Returns
Array of observables using rydiqule's stacking convention.

Return type
np.ndarray

get_phase_shift() → float | ndarray

Extract the phase shift from a solution.

Assumes the optically-thin approximation is valid.

Returns
Probe phase in radians.

Return type
float or numpy.ndarray

Examples

```
>>> [g, e] = rq.D2_states('Rb85')
>>> c = rq.Cell('Rb85', [g, e], cell_length =  0.00001)
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
(3,)
>>> print(sols.get_phase_shift())
80.5294887
```

get_solution_element(*idx: int*) → float | ndarray

Return a slice of an n_dimensional matrix of solutions of shape (…,n^2-1), where n is the basis size of the quantum system.

Parameters
idx (*int*) – Solution index to slice.

Returns
Slice of solutions corresponding to index idx. For example, if sols has shape (…, n^2-1), sol_slice will have shape (…).

Return type
float or numpy.ndarray

Raises
RydiquleError – If `idx` in not within the shape determined by basis size.

Examples

```
>>> [g, e] = rq.D2_states("Rb85", expand=True)
>>> c = rq.Cell('Rb85', [g, e])
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
(3,)
>>> rho_01_im = sols.get_solution_element(0)
>>> print(rho_01_im)
0.00131399
```

get_susceptibility() → complex | ndarray

Return the atomic susceptibility on the probe transition.

Experimental parameters must be defined manually for a `Sensor`.

Returns
Susceptibility of the density matrix solution.

Return type
complex or numpy.ndarray

Examples

```
>>> [g, e] = rq.D2_states(5, expand=True)
>>> c = rq.Cell('Rb85', [g, e], cell_length = 0.0001)
>>> c.add_coupling(states=(g,e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
(3,)
>>> sus = sols.get_susceptibility()
>>> print(sus)
(1.891136e-05+0.00036817j)
```

get_transmission_coef() → float | ndarray

Extract the transmission term from a solution.

Assumes the optically-thin approximation is valid.

Returns
Numerical value of the probe absorption in fractional units (P_out/P_in).

Return type
float or numpy.ndarray

Examples

```
>>> [g, e] = rq.D2_states('Rb85')
>>> c = rq.Cell('Rb85', [g, e], cell_length =  0.0001)
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho.shape)
(3,)
>>> t = sols.get_transmission_coef()
>>> print(t)
0.743427
```

init_cond: ndarray

Initial conditions, when using the time solver. Undefined otherwise.

Type
numpy.ndarray

property kappa: float

Kappa constant from the Cell. Not generally defined when using a Sensor.

property probe_freq: float

Probing transition frequency, in rad/s. Not generally defined when using a Sensor.

property probe_rabi: complex | ndarray

Base laser rabi frequency of the probing transition of the `Sensor` used in a solve.

The return of this function will be the appropriate shape to be cast using rydiqule's stacking convention. An error will be thrown if any of the base rabi frequencies of the probe coupling group do not match. (All rabi frequencies should match if added using `add_coupling_group` and not overwritten)

Returns

- *The base rabi frequency of the transition defined as the rabi frequency of any of the*
- couplings in the group divided by the `coherent_cc` on the corresponding edge (default 1).

property probe_tuple: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]]

Coupling edge corresponding to probing field. Defined as the first added coupling when using a Sensor.

rho: ndarray

Solutions returned by the solver.

Type

numpy.ndarray

rho_ij(*i: int | str | Tuple[float, ...], j: int | str | Tuple[float, ...]*) → complex | ndarray

Gets the full complex density matrix element corresponding to a given pair of indices.

Returns the entire array of density matrix elements corresponding to every combination of parameters defined by the mesh of parameters in the system. The shape of the output array will match the `stack_shape` of the solution.

Indices can be provided either as integers which number the states, or using state labels (integers, tuples, or strings). For the case of tuples, only individual states can be used, full state specifications are invalid

In the case of a state, it will be mapped to the corresponding integer, then the element itself is fetched using the `get_rho_ij()` function.

Parameters

- **i** (*int, tuple, or string*) – density matrix element row, or state label corresponding to row
- **j** (*int, tuple or string*) – density matrix element column, or state label corresponding to column

Returns

`[i,j]` complex element(s) of the density matrix

Return type

complex or numpy.ndarray

Examples

State labels and integer indices can be used interchangeably.

```
>>> rabis = np.linspace(1, 6, 5)
>>> my_sensor = rq.Sensor(["g","e"])
>>> my_sensor.add_coupling(("g","e"), rabi_frequency=rabis, detuning=1)
>>> my_sensor.add_decoherence(("e","g"), 0.1)
>>> sol = rq.solve_steady_state(my_sensor)
>>> print(sol.rho_ij("g","e"))
[0.3328-0.0166j 0.3184-0.0159j 0.2455-0.0123j 0.1933-0.0097j
 0.1579-0.0079j]
>>> print(sol.rho_ij(0,1))
[0.3328-0.0166j 0.3184-0.0159j 0.2455-0.0123j 0.1933-0.0097j
 0.1579-0.0079j]
```

rq_version: str

Version of rydiqule that created the Solution.

Type

str

property states: List[int | str | Tuple[float, ...]]

Return a list of all states in the `Sensor` used to produce solution.

Returns
List of all states in the sensor used to produce solution. Order will match order in `Sensor`.

Return type
list of states

t: ndarray

Times the solution is returned at, when using the time solver. Undefined otherwise.

Type
numpy.ndarray

variable_parameters: list

Output of the Sensor.variable_parameters() method for the Sensor used in a solve.

8.1.11 rydiqule.sensor_utils

Utilities used by the Sensor classes.

Functions

check_positive_semi_definite(dm)	Checks if the provided matrices is a physical density matrix.
convert_complex_to_dm(complex_dm)	Converts a standard density matrices in the complex basis with ground state into rydiqule's computational real basis with ground state removed.
convert_dm_to_complex(dm)	Converts density matrices from rydiqule's computational basis (real, with state 0 removed) to a standard complex basis with all states present.
convert_to_full_dm(dm)	Converts density matrices from rydiqule's computational basis (real, with state 0 removed) to the full, real basis (ie with state 0 population inserted).
coupling_subgraph(coupling, coupling_graph)	Returns a subgraph view of the `couplings_graph` corresponding to `coupling`.
draw_diagram(sensor[, include_dephasing])	Draw a matplotlib plot that shows the energy level diagram, couplings, and dephasing paths.
expand_statespec(statespec)	Returns a list of all possible `states` corresponding to a given `statespec`.
generate_eom(hamiltonian, gamma_matrix[, ...])	Create the optical bloch equations for a hamiltonian and decoherence matrix using the Lindblad master equation.
get_basis_transform(basis_size)	Function that defines the basis transformation matrix u and its inverse u_i, between the real and complex basis.
get_rho_ij(sols, i, j)	For a given density matrix solution, retrieve a specific element of the density matrix.
get_rho_populations(sols)	For a given density matrix solution, return the diagonal populations.
make_real(equations, constant[, ground_removed])	Converts equations of motion from complex basis to real basis.
match_states(statespec, compare_list)	Return all states in a list matching the pattern described by a given specification.
nx_edges_with(couplings_graph, *keys[, method])	Returns a version of couplings_graph with only the keys specified.
process_scannable_parameter(val)	Ensures that scannable parameters are coerced to numpy arrays.
remove_ground(equations)	Remove the ground state from the equations of motion using population conservation.

continues on next page

Table 8.38 – continued from previous page

scale_dipole(dipole)	Scale a dipole matrix from units of a0*e to Mrad/s when multiplied by a field in V/m.
state_tuple_to_str(states)	Helper function to create a more terse string representation of a tuple of state tuples.

rydiqule.sensor_utils.check_positive_semi_definite

rydiqule.sensor_utils.**check_positive_semi_definite**(*dm: ndarray*)

Checks if the provided matrices is a physical density matrix.

This is done by confirming each matrix of the stack is positive semi-definite.

Parameters

dm (*numpy.ndarray*) – Stack of density matrices in rydiqule's computational basis (i.e. real with ground state removed). Expected shape is `(..., N)` where `N = basis_size**2-1`.

Raises

RydiquleError – If at least one density matrix of the stack is not positive semi-definite.

rydiqule.sensor_utils.convert_complex_to_dm

rydiqule.sensor_utils.**convert_complex_to_dm**(*complex_dm: ndarray*) → ndarray

Converts a standard density matrices in the complex basis with ground state into rydiqule's computational real basis with ground state removed.

`rydiqule`'s built-in functions do not return complex density matrices, so this function is used
internally and to undo the conversion of *convert_dm_to_complex()*.

Parameters

complex_dm (*numpy.ndarray*) – Stack of density matrices in the complex basis with ground state present. Has shape of `(..., b, b)` where `b` is the number of states in the basis.

Returns

Density matrices in rydiqule's computational basis (real with state 0 removed). Has shape `(..., b**2-1)`.

Return type

numpy.ndarray

Raises

- ***RydiquleError*** – If the provided density matrices are not square.
- ***RydiquleError*** – If the converted matrix is not real (implies non-hermitian)

rydiqule.sensor_utils.convert_dm_to_complex

rydiqule.sensor_utils.**convert_dm_to_complex**(*dm: ndarray*) → ndarray

Converts density matrices from rydiqule's computational basis (real, with state 0 removed) to a standard complex basis with all states present.

Solutions computed using one of rydiqule's built in solvers will always output solutions in the real, 1-dimensional computational basis which is intended as the input for this function. Note that while a full complex density matrix can be useful to view solutions, performing calculations with the full complex density matrix can often add considerable unwanted rounding errors.

Parameters

dm (*numpy.ndarray*) – Density matrices in rydiqule's computational basis (real, with state 0 removed). Has shape `(..., b**2-1)` where `b` is the number of states in the basis.

Returns

Density matrices in the complex basis with state 0 present. Will have shape `(..., b, b)`.

Return type
numpy.ndarray

Raises
RydiquleError – If final dimension is of invalid size (i.e. does not correspond to b**2-1, where b is an integer)

Examples

```
>>> [g, e] = rq.D2_states('Rb85')
>>> c = rq.Cell('Rb85', [g, e], cell_length =  0.00001)
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho)
[0.001313 0.02558  0.000664]
>>> print(rq.sensor_utils.convert_dm_to_complex(sols.rho))
[[9.993359e-01+0.j       1.31399e-03+0.025581j]
 [1.313990e-03-0.025581j 6.64016e-04+0.j      ]]
```

rydiqule.sensor_utils.convert_to_full_dm

`rydiqule.sensor_utils.convert_to_full_dm(dm: ndarray) → ndarray`

Converts density matrices from rydiqule's computational basis (real, with state 0 removed) to the full, real basis (ie with state 0 population inserted).

Solutions computed using one of rydiqule's built in solvers will always output solutions in the real computational basis which is intended as the input for this function.

Parameters
dm (*numpy.ndarray*) – Density matrices in rydiqule's computational basis (real, with state 0 removed). Has shape `(..., b**2-1)` where `b` is the number of states in the basis.

Returns
Density matrices in the real basis with state 0 present. Will have shape `(..., b**2)`.

Return type
numpy.ndarray

Raises
RydiquleError – If final dimension is of invalid size (i.e. does not correspond to b**2-1, where b is an integer)

Examples

```
>>> [g, e] = rq.D2_states('Rb85')
>>> c = rq.Cell('Rb85', [g, e], cell_length =  0.00001)
>>> c.add_coupling(states=(g, e), rabi_frequency=1, detuning=1)
>>> sols = rq.solve_steady_state(c)
>>> print(sols.rho)
[0.001313 0.02558  0.000664]
>>> print(rq.sensor_utils.convert_to_full_dm(sols.rho))
[9.993359e-01 1.31399e-03 2.55811e-02 6.64016e-04]
```

rydiqule.sensor_utils.coupling_subgraph

`rydiqule.sensor_utils.coupling_subgraph(coupling: Tuple[int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]], coupling_graph: Graph) → Graph`

Returns a subgraph view of the `couplings_graph` corresponding to `coupling`.

Parameters

- **coupling** (*StateSpecs*) – Coupling specification
- **coupling_graph** (*networkx.Graph*) – Couplings graph to make the subgraph from

Returns
View of the corresponding subgraph

Return type
networkx.classes.graph.Graph

rydiqule.sensor_utils.draw_diagram

rydiqule.sensor_utils.**draw_diagram**(*sensor: Sensor*, *include_dephasing: bool = True*) → LD

Draw a matplotlib plot that shows the energy level diagram, couplings, and dephasing paths.

To show the plot, call `plt.show()`. If in a jupyter notebook, this is handled automatically.

Diagram has horizontal lines for the energy levels (spacing not to scale). Integer labels refer to the internal indexing for each state. If sensor is of type `Cell`, will also add text labels to each state of the quantum numbers.

Solid arrows between states are couplings defined with a non-zero Rabi frequency. Dashed arrows between states are couplings defined with a dipole moment.

Wiggly arrows between states denote a dephasing pathway. Opacity represents strength of dephasing relative to the largest specified dephasing, where fully opaque is the largest dephasing.

Parameters

- **sensor** (*Sensor*) – Sensor object to diagram.
- **include_dephasing** (*bool, optional*) – Whether to plot dephasing paths. Default is `True`.

Returns
Diagram handle

Return type
`leveldiagram.LD`

rydiqule.sensor_utils.expand_statespec

rydiqule.sensor_utils.**expand_statespec**(*statespec: int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...]*) → List[int | str | Tuple[float, ...]]

Returns a list of all possible `states` corresponding to a given `statespec`.

A `state` in `rydiqule` is defined by either a floating point or string value, or a tuple of such values. A `StateSpec` can replace any float or string value with a list of such values. The `expand_statespec` function's purpose is to convert a single `statespec` in which some number of elements are defined as lists into a list of all the states which correspond to the state.

If the provided spec is only a single state, a 1-element list containing that state is returned.

Parameters
statespec (*StateSpec*) – State specification with either zero or one element defined as a list.

Returns
List of all states matching the provided statespec

Return type
list of State

Raises
RydiquleError – If the provided `statespec` is not a valid state specification.

Notes

..note:

This function will preserve the state type for namedtuple statespecs. So for example, passing an `A_QState` for example will return a list of states of the same type.

Examples

```
>>> ground = (0,0)
>>> excited = (1, [0,1])
>>> print(rq.expand_statespec(ground))
[(0, 0)]
>>> print(rq.expand_statespec(excited))
[(1, 0), (1, 1)]
```

This function has utility in allowing otherwise cubersome state definitions to be defined with variables in `sensor.Sensor` functions.

```
>>> g = (0,0)
>>> e = (1, [-1,0,1])
>>> [em1, e0, ep1] = rq.expand_statespec(e)
>>> s = rq.Sensor([g,e])
>>> cc = {(g,em1): 0.25,
...       (g,e0): 0.5,
...       (g,ep1): 0.25}
>>> s.add_coupling((g,e), detuning=1, rabi_frequency=2, coupling_
↪coefficients=cc, label="probe")
>>> print(s.get_hamiltonian())
[[ 0.   +0.j  0.25+0.j  0.5 +0.j  0.25+0.j]
 [ 0.25-0.j -1.   +0.j  0.   +0.j  0.   +0.j]
 [ 0.5 -0.j  0.   +0.j -1.   +0.j  0.   +0.j]
 [ 0.25-0.j  0.   +0.j  0.   +0.j -1.   +0.j]]
```

rydiqule.sensor_utils.generate_eom

`rydiqule.sensor_utils.generate_eom`(*hamiltonian: ndarray*, *gamma_matrix: ndarray*, *remove_ground_state: bool = True*, *real_eom: bool = True*) → Tuple[ndarray, ndarray]

Create the optical bloch equations for a hamiltonian and decoherence matrix using the Lindblad master equation.

Parameters

- **hamiltonian** (*numpy.ndarray*) – Complex array representing the Hamiltonian matrix of the system, the matrix should be of shape `(*l, n, n)`, where n is the basis size and l is the shape of the stack of hamiltonians. For example, if the hamiltonian varies in 2 parameters l might be `(10, 10)`.
- **gamma_matrix** (*numpy.ndarray*) – Complex array representing the decoherence matrix of the system, the matrix should be of size `(n, n)`, where n is the basis size.
- **remove_ground_state** (*bool, optional*) – Remove the ground state from the equations of motion using population conservation. Setting to `False` is intended for internal use only and is not officially supported. See `remove_ground()` for details.
- **real_eom** (*bool, optional*) – Transform the equations of motion from the complex basis to the real basis. Setting to `False` is intended for internal use only and is not officially supported. See `make_real()` for details.

Returns

- **equations** (*numpy.ndarray*) – The array representing the Optical Bloch Equations (OBEs) of the system. The shape will be `(*l, n^2-1, n^2-1)` if `remove_ground_state` is `True` and `(*l, n^2, n^2)` otherwise. The datatype will be `np.float64` if `real_eom` is `True` and `np.complex128` otherwise.
- **const** (*numpy.ndarray*) – Array of which defines the constant term in the linear OBEs. The shape will be `(*l, n^2-1)` if `remove_ground_state` is `True` and `(*l, n^2)` otherwise. The datatype will be `np.float64` if `real_eom` is `True` and `np.complex128` otherwise.

Raises

`RydiquleError` – If the shapes of gamma_matrix and hamiltonian are not matching: or not square in the last 2 dimensions

Examples

```
>>> ham = np.diag([1,-1])
>>> gamma = np.array([[.1, 0],[.1,0]])
>>> print(ham.shape)
(2, 2)
>>> eom, const = rq.sensor_utils.generate_eom(ham, gamma)
>>> print(eom + 0.0)   # adding 0.0 prevents occasional -0.0 in doctest
[[-0.1  2.   0. ]
 [-2.  -0.1  0. ]
 [ 0.   0.  -0.1]]
>>> print(const.shape)
(3,)
```

This also works with a "stack" of multiple hamiltonians:

```
>>> ham_base = np.diag([1,-1])
>>> ham_full = np.array([ham_base for _ in range(10)])
>>> gamma = np.array([[.1, 0],[.1,0]])
>>> print(ham_full.shape)
(10, 2, 2)
>>> eom, const = rq.sensor_utils.generate_eom(ham_full, gamma)
>>> print(eom.shape)
(10, 3, 3)
>>> print(const.shape)
(10, 3)
```

rydiqule.sensor_utils.get_basis_transform

`rydiqule.sensor_utils.`**`get_basis_transform`**`(`*basis_size: int*`)` → Tuple[ndarray, ndarray]

Function that defines the basis transformation matrix u and its inverse u_i, between the real and complex basis.

This matrix u implements that the $\rho[j,i] \rightarrow Re(\rho[j,i])$ and $\rho[i,j] \rightarrow Im(\rho[j,i])$.

The transformation is not quite unitary, due to the asymmetry of the factors of 1/2.

Parameters

basis_size (*int*) – Size of the basis to generate transformations for.

Returns

- **u** (*numpy.ndarray*) – Forward transformation matrix.
- **u_inv** (*numpy.ndarray*) – Inverse transformation matrix.

Raises

`RydiquleError` – If `basis_size` does not match current basis.

rydiqule.sensor_utils.get_rho_ij

`rydiqule.sensor_utils.get_rho_ij(sols: ndarray | Solution, i: int, j: int) → complex | ndarray`

For a given density matrix solution, retrieve a specific element of the density matrix.

Assumes the ground state of the solution is eliminated (as per `remove_ground()`), and assumes Rydiqule's nominal state ordering of the Density Vector (per `make_real()`).

Parameters

- **sols** (numpy.ndarray or `Solution`) – Solutions to extract the matrix element for. Can be either the solution object returned by the solve or an N-D array representing density vectors, with ground state removed, and written in the totally real equations.
- **i** (`int`) – density matrix index i
- **j** (`int`) – density matrix index j

Returns

Array of rho_ij values. Will be of type float when `i==j`. Will be of type complex128 when `i!=j`.

Return type

numpy.ndarray

Examples

```
>>> sols = np.arange(180).reshape((4,5,3,3))
>>> print(sols.shape)
(4, 5, 3, 3)
>>> rho_01 = rq.get_rho_ij(sols, 0,1)
>>> print(rho_01.shape)
(4, 5, 3)
>>> print(rho_01[0,0])
[0.-1.j 3.-4.j 6.-7.j]
```

rydiqule.sensor_utils.get_rho_populations

`rydiqule.sensor_utils.get_rho_populations(sols: ndarray | Solution) → ndarray`

For a given density matrix solution, return the diagonal populations.

Note that rydiqule's convention for removing the ground state forces population conservation, ie the sum of these populations will be 1.

Parameters

sols (numpy.ndarray or `Solution`) – Solutions to extract the matrix element for. Can be either the solution object returned by the solve or an N-D array representing density vectors, with ground state removed, and written in the totally real equations.

Returns

Populations of the density matrices. Will have same shape as input solutions, with the last dimension reduced to the basis size.

Return type

numpy.ndarray

rydiqule.sensor_utils.make_real

`rydiqule.sensor_utils.make_real(equations: ndarray, constant: ndarray, ground_removed: bool = True) → Tuple[ndarray, ndarray]`

Converts equations of motion from complex basis to real basis.

Changes the density vector equation for p_ij into the Re[p_ij] equation and changing the density vector equation for p_ji into the equation for Im[p_ij].

Parameters

- **equations** (*numpy.ndarray*) – Complex equations of motion.
- **constant** (*numpy.ndarray*) – RHS of the equations of motion.
- **ground_removed** (*bool, optional*) – Indicates if `equations` has had the ground state removed. Default is `True`.

Returns

- **real_eqns** (*numpy.ndarray*) – EOMs in real basis.
- **real_const** (*numpy.ndarray*) – RHS of EOMs in real basis.

rydiqule.sensor_utils.match_states

rydiqule.sensor_utils.**match_states**(*statespec: int | str | Tuple[float, ...] | List[int | str | Tuple[float, ...]] | Tuple[float | List[float], ...], compare_list: List[int | str | Tuple[float, ...]]*) → List[int | str | Tuple[float, ...]]

Return all states in a list matching the pattern described by a given specification.

A `StateSpec` is described by a tuple containing floats or strings, or lists of floats or strings. A state `s` in `compare_list` is considered a match to `statespec` if `s` and `statespec` are the same length and for each element in `statespec`, :

1. **The corresponding element in `s` is equal to the element in `statespec` (in the case of a** single value)
2. **The element is `s` is an element of the list which is an element of `statespec` (in the** case of a list element of `statespec`).
3. The element of `statespec` is the string `"all"`

Parameters

- **statespec** (*StateSpec*) – The state specification against which to compare elements of the list.
- **compare_list** (*List[State]*) – The list of individual states to compare.

Returns

Sublist of `compare_list` containing all elements of `compare_list` matching `statespec`.

Return type

list of State

Examples

While primarily intended for internal use, `match_states` can be accessed directly through `sensor_utils`.

```
>>> compare_states = [(0,0),
...         (1,0),(1,1),
...         (2,0),(2,1),(2,2)
...         ]
>>> spec = (2,[0,1,2])
>>> print(rq.sensor_utils.match_states(spec, compare_states))
[(2, 0), (2, 1), (2, 2)]
>>> wildcard_spec = (2,"all")
>>> print(rq.sensor_utils.match_states(wildcard_spec, compare_states))
[(2, 0), (2, 1), (2, 2)]
```

rydiqule.sensor_utils.nx_edges_with

`rydiqule.sensor_utils.nx_edges_with(couplings_graph: Graph, *keys: str, method: Literal['all', 'any', 'not any', 'not all'] = 'all') → Dict[Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]], Dict]`

Returns a version of couplings_graph with only the keys specified.

Can be specified with a several criteria, including all, none, or any of the keys specified.

Parameters

- **keys** (*tuple of str*) – tuple of strings which should be one the valid parameter names for a state. See `add_coupling()` for which names are valid for a Sensor object.
- **method** (*Literal['all','any', 'not any', 'not all'], optional*) – Method to see if a given field matches the keys given. Choosing "all" will return couplings which have keys matching all of the values provided in the keys argument, while choosing "any", will return all couplings with keys matching at least one of the values specified by keys. For example, `sensor.couplings_with("rabi_frequency")` returns a dictionary of all couplings for which a rabi_frequency was specified. `sensor.couplings_with("rabi_frequency", "detuning", method="all")` returns all couplings for which both rabi_frequency and detuning are specified. 'sensor.couplings_with("rabi_frequency", "detuning", method="any")` returns all couplings for which either rabi_frequency or detuning are specified. "not any" and "not all" give the inverse of the above. Defaults to "all".

Returns

A copy of the `couplings_graph` edges dictionary with only couplings containing the specified parameter keys.

Return type

dict

rydiqule.sensor_utils.process_scannable_parameter

`rydiqule.sensor_utils.process_scannable_parameter(val: float | List[float] | ndarray) → ndarray | float`

Ensures that scannable parameters are coerced to numpy arrays.

If the parameter only has length of one, content is extracted.

Parameters

val (*ScannableParameter*) – Scannable parameter to process.

Return type

float or numpy.ndarray

Raises

RydiquleError: – Raised if passed an empty sequence

rydiqule.sensor_utils.remove_ground

`rydiqule.sensor_utils.remove_ground(equations: ndarray) → Tuple[ndarray, ndarray]`

Remove the ground state from the equations of motion using population conservation.

Population conservation enforces

$$\rho_{(0,0)} = 1 - \sum_{i=1}^{n-1} \rho_{(i,i)}$$

We use this equation to remove the EOM for rho_00 and enforce population conservation in the steady state.

Parameters

equations (*numpy.ndarray*) – array of shape (n^2, n^2) representing the equations of motion of the system, where n is the number of basis states.

Returns
The modified equations of shape (n^2-1, n^2-1)

Return type
numpy.ndarray

rydiqule.sensor_utils.scale_dipole

`rydiqule.sensor_utils.scale_dipole(dipole: float | ndarray) → float | ndarray`

Scale a dipole matrix from units of a0*e to Mrad/s when multiplied by a field in V/m.

Parameters
dipole (*float or numpy.ndarray*) – Array of dipole moments in units of a0*e. These are the default units used by ARC.

Returns
Scaled array in units of (Mrad/s)/(V/m)

Return type
numpy.ndarray

rydiqule.sensor_utils.state_tuple_to_str

`rydiqule.sensor_utils.state_tuple_to_str(states: Tuple[int | str | Tuple[float, ...], int | str | Tuple[float, ...]]) → str`

Helper function to create a more terse string representation of a tuple of state tuples.

The default python behavior for a str representation of tuples is to use `__repr__` for individual elements. We want to use `str`, since the output is pointlessly long otherwise. `A_QState.__repr__()` is longer than `A_QState.__str__()`

Parameters
states (*tuple of states*) – States for which to produce a string representation.

8.1.12 rydiqule.slicing

Modules

slicing	A handful of tools that solvers use to interface with slicing matrix stacks

rydiqule.slicing.slicing

A handful of tools that solvers use to interface with slicing matrix stacks

Functions

compute_grid(stack_shape, n_slices)	Calculate the bin edges to break a given stack shape into at least a certain number of pieces
get_slice_num(n, stack_shape, doppler_shape, ...)	Estimates the memory required for the desired steady state solve.
get_slice_num_hybrid(n, param_stack_shape, ...)	Estimates memory and determines the number of slices for the analytic solver.
get_slice_num_t(n, stack_shape, ...[, debug])	Estimates the memory required for the desired time solve.
matrix_slice(*matrices[, edges, n_slices])	Generator that returns parts of a stack of matrices.
memory_size(shape, item_size)	Calculate the memory size, in bytes of an array with the given size and shape.

rydiqule.slicing.slicing.compute_grid

`rydiqule.slicing.slicing.`**`compute_grid`**(*stack_shape: Tuple[int, ...], n_slices: int*) → List[ndarray]

Calculate the bin edges to break a given stack shape into at least a certain number of pieces

Works by iterating first over a number of slices per axis (N=1,2,3), then over each in the stack shape, splitting the axis into N slices, and comparing the total number of slices to the number specified. In a sense, the algorithm factors a number greater than or equal to n_slices, then breaks the stack along each axis according to this factorization. If the axis lengths do not break evenly into the appropriate number of pieces, the bin edges are truncated to an integer. This means that the slices are not guaranteed to be `(1/n_slices)`, but they will be close enough for most cases.

Parameters

- **`stack_shape`** (*`tuple of int`*) – The shape of the stack to be sliced. Does not include Hamiltonian or matrix equation dimensions, so for a hamiltonain stack of shape `(*l,n, n)`, `stack_shape` will be `*l`.
- **`n_slices`** (*`int`*) – The number of slices into which to break the hamiltonian. Lower bound on the number of slices there will actually be.

Returns

The list of bin edges axis by axis. Can be passed to *`matrix_slice()`* as the `edges` argument.

Return type

list(np.ndarray)

Examples

```
>>> import rydiqule.slicing.slicing as slicing
>>> stack_shape=(10,10)
>>> print(slicing.compute_grid(stack_shape, 4))
[array([ 0,  5, 10]), array([ 0,  5, 10])]
>>> print(slicing.compute_grid(stack_shape, 6))
[array([ 0,  3,  6, 10]), array([ 0,  5, 10])]
```

rydiqule.slicing.slicing.get_slice_num

`rydiqule.slicing.slicing.`**`get_slice_num`**(*n: int, stack_shape: Tuple[int, ...], doppler_shape: Tuple[int, ...], sum_doppler: bool, weight_doppler: bool, n_slices: int | None = None, debug: bool = False*) → Tuple[int, Tuple[int, ...]]

Estimates the memory required for the desired steady state solve.

Estimates are fairly accurate, but not guaranteed. Goal is to err on allowing edge case solves to proceed.

Parameters

- **`n`** (*`int`*) – Size of the system basis
- **`stack_shape`** (*`tuple of int`*) – Tuple of sizes for the hamiltonian stack to be solved
- **`doppler_shape`** (*`tuple of int`*) – Tuple of sizes for the doppler axes. Pass an empty tuple if no doppler averaging.
- **`sum_doppler`** (*`bool`*) – Whether solution will be summing the doppler average
- **`weight_doppler`** (*`bool`*) – Whether the solution will apply weights to the doppler averaging
- **`n_slices`** (*`int, default=1`*) – Manually override the minimum number of hamiltonian slices to use.
- **`debug`** (*`bool, default=False`*) – Print debug information about the memory calculations.

Returns

- **n_ham_slices** (*int*) – Number of slices to use when solving the stacked hamiltonian
- **out_sol_shape** (*tuple of int*) – Shape of the resulting solution for this calculation.

Raises

- `RydiquleError` – If there isn't enough memory to solve the system:
- `RydiquleError` – If `sum_doppler=False` and full solution does not fit in memory.:

rydiqule.slicing.slicing.get_slice_num_hybrid

`rydiqule.slicing.slicing.get_slice_num_hybrid`(*n: int*, *param_stack_shape: Tuple[int, ...]*, *numeric_doppler_shape: Tuple[int, ...]*, *n_slices: int | None = None*, *debug: bool = False*) → Tuple[int, Tuple[int, ...]]

Estimates memory and determines the number of slices for the analytic solver.

This version is tailored to the memory footprint of the analytic algorithm, which includes large intermediate arrays like the propagator and eigenvector stacks.

Parameters

- **n** (`int`) – Size of the system basis.
- **param_stack_shape** (`tuple of int`) – Tuple of sizes for the sensor's parameter axes (e.g., from L0.shape[:-2]).
- **numeric_doppler_shape** (`tuple of int`) – Tuple of sizes for the numeric doppler axes. Pass an empty tuple for the 1D case.
- **n_slices** (`int, optional`) – Manually override the minimum number of slices. If None, it's determined automatically.
- **debug** (`bool, optional`) – If True, prints detailed memory usage information.

Returns

- **n_param_slices** (*int*) – Number of slices to use when iterating over the parameter stack.
- **out_sol_shape** (*tuple of int*) – Shape of the final, fully-solved solution array.

rydiqule.slicing.slicing.get_slice_num_t

`rydiqule.slicing.slicing.get_slice_num_t`(*n: int*, *stack_shape: Tuple[int, ...]*, *doppler_shape: Tuple[int, ...]*, *time_points: int*, *sum_doppler: bool*, *weight_doppler: bool*, *n_slices: int | None*, *debug: bool = False*) → Tuple[int, Tuple[int, ...]]

Estimates the memory required for the desired time solve.

Note that the time solver used (scipy.solve_ivp) is an adaptive solver, so the internal number of time steps used is problem dependent and not controlled by the requested number of time points. Generally, the number of points is proportional to the highest frequency in the problem and the length of the time to solve. To estimate a lower bound on the memory needed to time solve, we use a fudge factor of 4 on the requested time points. This is unlikely to be accurate even in a general case.

Parameters

- **n** (`int`) – Size of the system basis
- **stack_shape** (`tuple of int`) – Tuple of sizes for the hamiltonian stack to be solved
- **doppler_shape** (`tuple of int`) – Tuple of sizes for the doppler axes. Pass an empty tuple if no doppler averaging.

- **time_points** (*int*) – Number of time steps requested from the time solver. This sets the output solution shape. An internal fudge factor of 4 is applied for memory estimation purposes.
- **sum_doppler** (*bool*) – Whether solution will be summing the doppler average
- **weight_doppler** (*bool*) – Whether the solution will apply weights to the doppler averaging
- **n_slices** (*int, default=1*) – Manually override the minimum number of hamiltonian slices to use.
- **debug** (*bool, default=False*) – Print debug information about the memory calculations.

Returns

- **n_ham_slices** (*int*) – Number of slices to use when solving the stacked hamiltonian
- **out_sol_shape** (*tuple of int*) – Shape of the resulting solution for this calculation.

Raises

RydiquleError – If `sum_doppler=False` and full solution does not fit in memory.:

Warns

RydiquleWarning (*If there is unlikely to be enough memory to solve the system.*)

rydiqule.slicing.slicing.matrix_slice

`rydiqule.slicing.slicing.`**`matrix_slice`**`(`**matrices: ndarray, edges: Iterable | None = None, n_slices: int | None = None*`)` → Iterator[Tuple[Tuple[slice, ...], Unpack[Tuple[ndarray, ...]]]]

Generator that returns parts of a stack of matrices.

Given a stack of n by n matrices, produces a genererator which returns the given matrices in the specified number of smaller stacks. For example, given a stack of matrices of shape `(10,10,4,4)` with 4 slices, generates 4 stacks of shape `(5,4,4)`. Due to the nature of the slicing, the number of slices might be slightly greater that the number specified. Output matrices will be broadcastable by numpy's broadcasting rules. Input arrays are interpreted as a stack, with the last 2 dimensions staying intact.

Parameters

- **matrices** (*np.ndarray*) – matrix stacks to be sliced. All matrices must be of shapes that can be broadcast by numpy's broadcasting rules, with the additional restriction that all matrices must have the same number of dimensions, even if some dimensions are of size 1. For example, matrices of sizes `(10,1,4,4)` and `(1,20,4,4)` can be sliced together.
- **edges** (*iterable, optional*) – The values along each axis that define the edges of bins on an n-dimensional grid. For example, to slice a grid of hamiltonians with stack_shape `(10,10)` into 4 pieces, `edges` could be defined as `[0,5,10]` for each of the 2 stack axes.
- **n_slices** (*int, optional*) – The number of slices into which to break the matrix stack. Ignored if the `edges` parameter is not `None`. Must be specified as an integer value if `edges` is `None`, ignored otherwise. Defaults to None.

Yields

- *tuple of slices* – slicing for each corresponding matrix
- *numpy.ndarray* – Slice of hamiltonian stack

Examples

```
>>> M1 = np.ones((1,20,4,4))
>>> M2 = np.ones((20,1,4,4))
>>> M3 = np.ones((20,20,4,4))
>>> axis0_edges = np.array([0,10,20])
>>> axis1_edges = np.array([0,10,20])
>>> for idx,m1,m2, m3 in rq.slicing.slicing.matrix_slice(M1, M2, M3,␣
→edges=[axis0_edges, axis1_edges]):
...     print(m1.shape, m2.shape, m3.shape)
(1, 10, 4, 4) (10, 1, 4, 4) (10, 10, 4, 4)
(1, 10, 4, 4) (10, 1, 4, 4) (10, 10, 4, 4)
(1, 10, 4, 4) (10, 1, 4, 4) (10, 10, 4, 4)
(1, 10, 4, 4) (10, 1, 4, 4) (10, 10, 4, 4)
```

rydiqule.slicing.slicing.memory_size

rydiqule.slicing.slicing.**memory_size**(*shape: Tuple[int, ...]*, *item_size: int*) → int

Calculate the memory size, in bytes of an array with the given size and shape. Does not calculate the actual array, just theoretical size since this function is intended to be used before attempting allocate an array that is too large.

Parameters

- **shape** (*list-like*) – Shape of the array in question.
- **item_size** (*int*) – Size of an array element in bytes.

Returns

Expected memory size of array in bytes

Return type

int

8.1.13 rydiqule.solvers

Steady-state solvers of the Optical Bloch Equations.

Functions

solve_steady_state(sensor[, doppler, ...])	Finds the steady state solution for a system characterized by a sensor.
steady_state_solve_stack(eom, const)	Helper function which returns the solution to the given equations of motion

rydiqule.solvers.solve_steady_state

rydiqule.solvers.**solve_steady_state**(*sensor: Sensor*, *doppler: bool = False*, *doppler_mesh_method: UniformMethod | IsoPopMethod | SplitMethod | DirectMethod | None = None*, *sum_doppler: bool = True*, *weight_doppler: bool = True*, *n_slices: int | None = None*) → *Solution*

Finds the steady state solution for a system characterized by a sensor.

If insufficient system memory is available to solve the system in a single call, system is broken into “slices” of manageable memory footprint which are solved individually. This slicing behavior does not affect the result. Can be performed with or without doppler averaging.

Parameters

- **sensor** (*Sensor*) – The sensor for which the solution will be calculated.

- **doppler** (*bool*, *optional*) – Whether to calculate the solution for a doppler-broadened gas. If `True`, only uses doppler broadening defined by `kvec` parameters for couplings in the `sensor`, so setting this `True` without `kvec` definitions will have no effect. Default is `False`.
- **(dict** (*doppler_mesh_method*) – If not `None`, should be a dictionary of meshing parameters to be passed to *doppler_classes()*. See *doppler_classes()* for more information on supported methods and arguments. If `None, uses the default doppler meshing. Default is `None.`
- **optional)** – If not `None`, should be a dictionary of meshing parameters to be passed to *doppler_classes()*. See *doppler_classes()* for more information on supported methods and arguments. If `None, uses the default doppler meshing. Default is `None.`
- **sum_doppler** (*bool*) – Whether to average over doppler classes after the solve is complete. Setting to `False` will not perform the sum, allowing viewing of the weighted results of the solve for each doppler class. In this case, an axis will be prepended to the solution for each axis along which doppler broadening is computed. Ignored if `doppler=False`. Default is `True`.
- **weight_doppler** (*bool*) – Whether to apply weights to doppler solution to perform averaging. If `False`, will **not** apply weights or perform a doppler_average, regardless of the value of `sum_doppler`. Changing from default intended only for internal use. Ignored if `doppler=False` or `sum_doppler=False`. Default is `True`.
- **n_slices** (*int or None*, *optional*) – How many sets of equations to break the full equations into. The actual number of slices will be the largest between this value and the minimum number of slices to solve the system without a memory error. If `None`, uses the minimum number of slices to solve the system without a memory error. Detailed information about slicing behavior can be found in *matrix_slice()*. Default is `None`.

Notes

Note

If decoherence values are not sufficiently populated in the sensor, the resulting equations may be singular, resulting in an error in `numpy.linalg`. This error is not caught for flexibility, but is likely the culprit for `numpy.linalg` errors encountered in steady-state solves.

Note

The solution produced by this function will be expressed using rydiqule's convention of converting a density matrix into the real basis and removing the ground state to improve numerical stability.

Note

If the sensor contains couplings with `time_dependence`, this solver will add those couplings at their $t = 0$ value to the steady-state hamiltonian to solve.

Returns

An object containing the solution and related information.

Return type

Solution

Examples

A basic solve for a 3-level system would have a "density matrix" solution of size 8 (3^2-1)

```
>>> s = rq.Sensor(3)
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(type(sol))
<class 'rydiqule.sensor_solution.Solution'>
>>> print(type(sol.rho))
<class 'numpy.ndarray'>
>>> print(sol.rho.shape)
(8,)
```

Defining an array-like parameter will automatically calculate the density matrix solution for every value. Here we use 11 values, resulting in 11 density matrices. The `axis_labels` attribute of the solution can clarify which axes are which.

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(sol.rho.shape)
(11, 8)
>>> print(sol.axis_labels)
['(0,1)_detuning', 'density_matrix']
```

```
>>> s = rq.Sensor(3)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = det, rabi_frequency=2)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s)
>>> print(sol.rho.shape)
(11, 11, 8)
>>> print(sol.axis_labels)
['(0,1)_detuning', '(1,2)_detuning', 'density_matrix']
```

If the solve uses doppler broadening, but not averaging for doppler is specified, there will be a solution axis corresponding to doppler classes.

```
>>> s = rq.Sensor(3, vP=1)
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, kvec=(4,0,0))
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_steady_state(s, doppler=True, sum_doppler=False)
>>> print(sol.rho.shape)
(561, 11, 8)
>>> print(sol.axis_labels)
['doppler_0', '(0,1)_detuning', 'density_matrix']
```

rydiqule.solvers.steady_state_solve_stack

rydiqule.solvers.**steady_state_solve_stack**(*eom: ndarray*, *const: ndarray*) → ndarray

Helper function which returns the solution to the given equations of motion

Solves an equation of the form $\dot{x} = Ax + b$, or a set of such equations arranged into stacks. Essentially just wraps numpy.linalg.solve(), but included as its own function for modularity if another solver is found to be worth investigating.

Parameters

- **eom** (*numpy.ndarray*) – An square array of shape `(*l,n,n)` representing the differential equations to be solved. The matrix (or matrices) A in the above formula.
- **const** (*numpy.ndarray*) – An array or shape `(*l,n)` representing the constant in the matrix form of the differential equation. The constant b in the above formula. Stack shape `*l` must be consistent with that in the `eom` argument

Returns

A 1xn array representing the steady-state solution of the differential equation

Return type

numpy.ndarray

8.1.14 rydiqule.stack_solvers

Modules

cyrk_solver
scipy_solver

rydiqule.stack_solvers.cyrk_solver

Functions

cyrk_solve(eoms_base, const_base, ...[, eqns])	Solve a set of Optical Bloch Equations (OBEs) with rydiqule's time solving convention using CyRK's `pysolve_ivp`.

rydiqule.stack_solvers.cyrk_solver.cyrk_solve

rydiqule.stack_solvers.cyrk_solver.**cyrk_solve**(*eoms_base: ndarray*, *const_base: ndarray*, *eom_time_r: ndarray*, *const_r: ndarray*, *eom_time_i: ndarray*, *const_i: ndarray*, *time_inputs: Sequence[Callable[[float], complex]]*, *t_eval: ndarray*, *init_cond: ndarray*, *eqns: Literal['orig', 'flat'] = 'orig'*, ***kwargs*) → ndarray

Solve a set of Optical Bloch Equations (OBEs) with rydiqule's time solving convention using CyRK's `pysolve_ivp`.

Uses matrix components of the equations of motion provided by the methods of a `Sensor()`. Designed to be used as a wrapped function within *solve_time()*. Builds and solves equations of motion according rydiqule's time solving conventions. Sets up and solves dx/dt = A(t)x + b(t)

For larger solve systems, `max_ram_MB` kwarg for `pysolve_ivp` will likely need to be increased from its default of 2000.

Parameters

- **eoms_base** (*numpy.ndarray*) – The matrix of shape `(*l,n,n)` representing the non time-varying portion of the matrix A in the equations of motion.

- **const_base** (*numpy.ndarray*) – The array of shape (*l, n) representing the non time-varying portion of the vector b in the equations of motion.
- **eoms_time_r** (*numpy.ndarray*) – The matrix of shape (n_t, *l, n, n) representing the real time-varying portion of the matrix A, where n_t is the length of time_inputs. The ith slice along the first axis should be multiplied by the real part of the ith entry in time_inputs.
- **const_r** (*numpy.nd_array*) – The matrix of shape (n_t, *l, n) representing the real time-varying portion of the vector b, where n_t is the length of time_inputs. The ith slice along the first axis should be multiplied by the real part of the ith entry in time_inputs.
- **eoms_time_i** (*numpy.ndarray*) – The matrix of shape (n_t, *l, n, n) representing the imaginary time-varying portion of the matrix A, where n_t is the length of time_inputs. The ith slice along the first axis should be multiplied by the imaginary part of the ith entry in time_inputs.
- **const_i** (*numpy.nd_array*) – The matrix of shape (n_t, *l, n) representing the imaginary time-varying portion of the vector b, where n_t is the length of time_inputs. The ith slice along the first axis should be multiplied by the imaginary part of the ith entry in time_inputs.
- **time_inputs** (*list (callable)*) – List of callable functions of length n_t. The functions should take a single floating point as an input representing the time in microseconds, and return a real or complex floating point value represent an electric field in V/m at that time. Return type of each function must be the same for all inputs t.
- **t_eval** (*numpy.ndarray*) – Array of times to sample the integration at. This array must have dtype of float64.
- **init_cond** (*numpy.ndarray*) – Matrix of shape (*l, n) representing the initial state of the system.
- **eqns** (*{"orig", "flat"}*) – Method used to generate the derivative equations. Options are orig (which uses a numpy reshaping approach) and flat (which uses flat array indexing).
- ****kwargs** (*dict:*) – Additional keyword arguments passed to pysolve_ivp.

Returns

The matrix solution of shape (*l,n,n_t) representing the density matrix of the system at each time t.

Return type

numpy.ndarray

Raises

RydiquleError – If system size exceeds cyrk backend limit of 65535 equations. If we see this error a lot, consider getting CyRK project to increase it by changing type of y_size from unsigned short.

rydiqule.stack_solvers.scipy_solver

Functions

scipy_solve(eoms_base, const, eom_time_r, ...)	Solve a set of Optical Bloch Equations (OBEs) with rydiqule's time solving convention using scipy's solve_ivp.

rydiqule.stack_solvers.scipy_solver.scipy_solve

rydiqule.stack_solvers.scipy_solver.**scipy_solve**(*eoms_base: ndarray*, *const: ndarray*, *eom_time_r: ndarray*, *const_r: ndarray*, *eom_time_i: ndarray*, *const_i: ndarray*, *time_inputs: Sequence[Callable[[float], complex]]*, *t_eval: ndarray*, *init_cond: ndarray*, *eqns: Literal['loop', 'comp'] = 'loop'*, ***kwargs*) → ndarray

Solve a set of Optical Bloch Equations (OBEs) with rydiqule's time solving convention using scipy's `solve_ivp`.

Uses matrix components of the equations of motion provided by the methods of a `Sensor()`. Designed to be used as a wrapped function within `solve_time()`. Builds and solves equations of motion according rydiqule's time solving conventions. Sets up and solves dx/dt = A(t)x + b(t)

Parameters

- **eoms_base** (*numpy.ndarray*) – The matrix of shape `(*l,n,n)` representing the non time-varying portion of the matrix A in the equations of motion.
- **const** (*numpy.ndarray*) – The array of shape `(*l, n)` representing the non time-varying portion of the vector b in the equations of motion.
- **eoms_time_r** (*numpy.ndarray*) – The matrix of shape `(n_t, *l, n, n)` representing the real time-varying portion of the matrix A, where n_t is the length of `time_inputs`. The ith slice along the first axis should be multiplied by the real part of the ith entry in `time_inputs`.
- **const_r** (*numpy.nd_array*) – The matrix of shape `(n_t, *l, n)` representing the real time-varying portion of the vector b, where n_t is the length of `time_inputs`. The ith slice along the first axis should be multiplied by the real part of the ith entry in `time_inputs`.
- **eoms_time_i** (*numpy.ndarray*) – The matrix of shape `(n_t, *l, n, n)` representing the imaginary time-varying portion of the matrix A, where n_t is the length of `time_inputs`. The ith slice along the first axis should be multiplied by the imaginary part of the ith entry in `time_inputs`.
- **const_i** (*numpy.nd_array*) – The matrix of shape `(n_t, *l, n)` representing the imaginary time-varying portion of the vector b, where n_t is the length of `time_inputs`. The ith slice along the first axis should be multiplied by the imaginary part of the ith entry in `time_inputs`.
- **time_inputs** (*list(callable)*) – List of callable functions of length `n_t`. The functions should take a single floating point as an input representing the time in microseconds, and return a real or complex floating point value represent an electric field in V/m at that time.
- **t_eval** (*numpy.ndarray*) – Array of times to sample the integration at.
- **init_cond** (*(numpy.ndarray)*) – Matrix of shape `(*l, n)` representing the initial state of the system.
- **eqns** (*{"loop", "comp"}*) – Function used of generating equations of motion. One of "loop" or "comp", corresponding to defining time-dependent equations of motion as a loop over time-dependent components or with a list comprehension. List comprehensions are preferred for longer solves and loops are preferred for shorter solves.
- ****kwargs** (*dict*) – Additional keyword arguments passed to the nbkode solver constructor.

Returns
The matrix solution of shape `(*l,n,n_t)` representing the density matrix of the system at each time t.

Return type
numpy.ndarray

8.1.15 rydiqule.timesolvers

Solvers for time domain analysis with an arbitrary RF field

Functions

generate_eom_time(hamiltonians_time)	Generates the Optical Bloch Equations for just the rf terms.
solve_eom_stack(eoms_base, const, ...)	Solve a stack of equations of motion with shape `(*l, n, n)` in the time domain.
solve_time(sensor, end_time, num_pts[, ...])	Solves the response of the optical sensor in the time domain given the its time-dependent inputs

rydiqule.timesolvers.generate_eom_time

`rydiqule.timesolvers.`**`generate_eom_time`**(*hamiltonians_time: ndarray*) → Tuple[ndarray, ndarray]

Generates the Optical Bloch Equations for just the rf terms. Uses the convention of the hamiltonian_rf return of the get_time_hamiltonian function. The equations of motion returned are assumed to be used in conjunction with an electric field.

Parameters
`hamiltonians_time` (*numpy.ndarray*) – A matrix of shape (basis_size, basis_size), where the off-diagonal terms (i,j) are the dipole matrix elements in e a_b of the transition coupling state i to state j.

Returns

- **numpy.ndarray** (*Part of the Optical Bloch Equations corresponding to time_dependent couplings.*) – To produce equations to solve, these values must be multiplied by an electric field in V/m.
- **numpy.ndarray** (*Constant term of the time-dependent portion of the equations*) – of motion. Same units as the equations themselves.

rydiqule.timesolvers.solve_eom_stack

`rydiqule.timesolvers.`**`solve_eom_stack`**(*eoms_base: ndarray*, *const: ndarray*, *eom_time_r: ndarray*, *const_r: ndarray*, *eom_time_i: ndarray*, *const_i: ndarray*, *time_inputs: List[Callable[[float], complex]]*, *t_eval: ndarray*, *init_cond: ndarray*, *solver*, ***kwargs*) → ndarray

Solve a stack of equations of motion with shape `(*l, n, n)` in the time domain.

Companion function to `solve_time()`, but can be invoked on its for equations already formatted.

Parameters

- **`eoms_base`** (*numpy.ndarray*) – Array of shape `(*l, n, n)` representing the part of equations of motion of the system which do not respond to external fields.
- **`const`** (*numpy.ndarray*) – constant term of shape (n,) added in differential equations. Typically generated by `generate_eom()`.
- **`eoms_time_r`** (*list[numpy.ndarray]*) – list of arrays of shape `(basis_size^2-1, basis_size^2-1)` representing the parts of the OBEs with a real-valued time-dependence. In the solver, this array will be multiplied by a

time-dependent rabi frequency. Typically a matrix of mostly zeros, with non-zero terms corresponding to a particular time-dependent coupling

- **const_r** (*numpy.ndarray*) – Constant term of shape (n,) added in a real time-dependent portion of differential equations. Typically generated by `generate_eom_time()`.
- **eoms_time_i** (*numpy.ndarray*) – list of arrays of shape `(basis_size^2-1, basis_size^2-1)` representing the parts of the OBEs with an imaginary-valued time-dependence. In the solver, this array will be multiplied by a time-dependent rabi frequency.
- **const_i** (*numpy.ndarray*) – constant term of shape (n,) added in an imaginary time-dependent portion of differential equations. Typically generated by `generate_eom_time()`.
- **t_eval** (*numpy.ndarray*) – 1-D array of times, in microseconds, at which to evaluate the solution. Does not affect evaluations in the solve.
- **time_inputs** (*list[function float->float]*) – List of functions which represent the rabi frequency of a field as a function of time. list length should be identical to the length of obes_time. In the solver, the i th time input will be evaluated at time t and multiplied by the i th entry of obes_time.
- **tuple(float)** (*time_range*) – Pair of values represent the start and end time, in microseconds, of the simulation.
- **init_cond** (numpy.ndarray or `None`, optional) – Density matrix representing the initial state of the system. If specified, the shape should be either `(n)` in the case of a single initial condition for all parameter values, or should be of shape `(*l, n)` matching the output shape of a steady state solve if the initial condition may be different for different combinations of parameters. If `None`, will solve the problem in the steady state with all time-dependent fields "off" and use the solution as the initial condition for the time behavior. Other possible manual options might include a matrix populated by zeros representing the entire population in the ground state. Defaults to `None`.

Returns
Flattened solution array corresponding to time points.

Return type
numpy.ndarray

rydiqule.timesolvers.solve_time

`rydiqule.timesolvers.`**`solve_time`**(*sensor: Sensor*, *end_time: float*, *num_pts: int*, *init_cond: ndarray | None = None*, *doppler: bool = False*, *doppler_mesh_method: UniformMethod | IsoPopMethod | SplitMethod | DirectMethod | None = None*, *sum_doppler: bool = True*, *weight_doppler: bool = True*, *n_slices: int | None = None*, *solver: Callable | Literal['scipy', 'cyrk'] = 'scipy'*, ***kwargs*) → *Solution*

Solves the response of the optical sensor in the time domain given the its time-dependent inputs

If insufficient system memory is available to solve the system all at once, system is broken into "slices" of manageable memory footprint which are solved individually. This slicing behavior does not affect the result. All couplings that include a "time_dependence" argument will be solved in the time domain.

A number of solver backends work with rydiqule, but the default `"scipy"` ivp solver is the is recommended backend in almost all cases, as it is the most fully-featured and documented. Advanced users have the ability to define their own solver backends by creating a function that follows the call signature for rydiqule timesolver backends. Additional arguments to the solver backend can be supplied with `**kwargs`.

Parameters

- **sensor** (*Sensor*) – The sensor object representing the atomic/laser arrangement of the system.

- **end_time** (*float*) – Amount of time, in microseconds, for which to simulate the system
- **num_pts** (*int*) – The number of points along the range `(0, end_time)` for which the solution is evaluated. This does not affect the number of function evaluations during the solve, rather the spacing of the points in the reported solution.
- **init_cond** (numpy.ndarray or `None`, optional) – Density matrix representing the initial state of the system. If specified, the shape should be either `(n)` in the case of a single initial condition for all parameter values, or should be of shape `(*l, n)` matching the output shape of a steady state solve if the initial condition may be different for different combinations of parameters. If `None`, will solve the problem in the steady state with all time-dependent fields at their $t = 0$ value and use the solution as the initial condition. Other possible manual options might include a matrix populated by zeros representing the entire population in the ground state. Defaults to `None`.
- **doppler** (*bool, optional*) – Whether to account for doppler shift among moving atoms in the gas. If True, the solver will implicitly define a velocity distribution for particles in the cell, solve the problem for each velocity class, and return a weighted average of the results. Note that solving in this manner carries a substantial performance penalty, as each doppler velocity class is solved as its own problem. If solved with doppler, only axis specified by a `"kvec"' argument in one of the sensor couplings will be average over. The time solver currently supports doppler averaging in any number of spatial dimensions, up to the limit of 3 imposed by the macroscopic physical world. Defaults to `False`.
- **doppler_mesh_method** (*dict, optional*) – Dictionary that controls the doppler meshing method. Exact details of this are found in the documentation of `doppler_classes()`. Ignored if `doppler=False`. Default is `None`.
- **sum_doppler** (*bool, optional*) – Whether to average over doppler classes after the solve is complete. Setting to false will not perform the sum, allowing viewing of the weighted results of the solve for each doppler class. Ignored if `doppler=False`. Default is `True`.
- **weight_doppler** (*bool*) – Whether to apply weights to doppler solution to perform averaging. If `False`, will **not** apply weights or perform a doppler_average, regardless of the value of `sum_doppler`. Changing from default intended only for internal use. Ignored if `doppler=False` or `sum_doppler=False`. Default is `True`.
- **n_slices** (*int or None, optional*) – How many sets of equations to break the full equations into. The actual number of slices will be the largest between this value and the minimum number of slices to solve the system without a memory error. If `None`, solver uses the minimum number of slices required to solve without a `memoryError`. Defaults to None.
- **solver** (*{"scipy", "cyrk"} or callable*) – The backend solver used to solve the ivp generated by the sensor. All string values correspond to backend solvers built in to rydiqule. Valid string values are:
  - "scipy": Solves equations with `scipy.integrate.solve_ivp()`. The default, most stable, and well-supported option.
  - "cyrk": Solves jit-compiled equations with a cython compiled RK solver from `CyRK`. Due to some jit compilation, only faster for moderate length problems (ie problems with a moderate number of required time steps).

  Additionally, can be specified with a callable that matches rydiqule's time-solver convention, enabling using a custom solver backend.

Unless otherwise noted, backends other than scipy are considered experimental. Issues with their use are considered features not fully implemented rather than bugs.

- ****kwargs** (*Additional keyword arguments passed to the backend solver.*) – See documentation of the relevant solver (i.e. `scipy.integrate.solve_ivp()`) for details and supported arguments.

Returns
An object containing the solution and related information. Timesolver-specific defined attributes are `t` and `init_cond`, corresponding respectively to the times at which the solution is sampled and the initial conditions used for the solve.

Return type
Solution

Examples

A basic solve for a 3-level system would have a "density matrix" solution of size 8 (3^2-1). Here we use a trivial time dependence for demonstration purposes, but in practice the time dependence is likely more complicated. Below the most basic use of `solve_time` is demonstrated

```
>>> s = rq.Sensor(3)
>>> td = lambda t: 1
>>> s.add_coupling((0,1), detuning = 1, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, time_dependence=td)
>>> s.add_transit_broadening(0.1)
>>> end_time = 10 #microseconds
>>> n_pts = 1000 #interpolated points in solution
>>> sol = rq.solve_time(s, end_time, n_pts)
>>> print(type(sol))
<class 'rydiqule.sensor_solution.Solution'>
>>> print(type(sol.rho))
<class 'numpy.ndarray'>
>>> print(sol.rho.shape)
(1000, 8)
```

Defining an array-like parameter will automatically calculate the density matrix solution for every value. Here we use 11 values, resulting in 11 density matrices. The `axis_labels` attribute of the solution can clarify which axes are which.

```
>>> s = rq.Sensor(3)
>>> td = lambda t: 1
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, time_dependence=td)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_time(s, end_time, n_pts)
>>> print(sol.rho.shape)
(11, 1000, 8)
>>> print(sol.axis_labels)
['(0,1)_detuning', 'time', 'density_matrix']
```

As expected, multiple axes of scanned parameters are handled the same way as they are in the steady-state case, with the expected additions from the time solver.

```
>>> s = rq.Sensor(3)
>>> td = lambda t: 1
>>> det = np.linspace(-1,1,11)
```

(continues on next page)

(continued from previous page)

```
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1)
>>> s.add_coupling((1,2), detuning = det, rabi_frequency=2, time_dependence=td)
>>> s.add_transit_broadening(0.1)
>>> sol = rq.solve_time(s, end_time, n_pts)
>>> print(sol.rho.shape)
(11, 11, 1000, 8)
>>> print(sol.axis_labels)
['(0,1)_detuning', '(1,2)_detuning', 'time', 'density_matrix']
```

If the solve uses doppler broadening, all doppler classes will be computed a weighted average will be taken over the doppler axis, and the shape of the solution will not change. While this is the desired behavior in most situations, the `sum_doppler` argument can be used to override this behavior, leave (or more) solution axis corresponding to different doppler classes.

```
>>> s = rq.Sensor(3, vP=1)
>>> td = lambda t: 1
>>> det = np.linspace(-1,1,11)
>>> s.add_coupling((0,1), detuning = det, rabi_frequency=1, kvec=(4,0,0))
>>> s.add_coupling((1,2), detuning = 2, rabi_frequency=2, kvec=(-4,0,0), time_
↪dependence=td)
>>> s.add_transit_broadening(0.1)
>>> end_time = 10 #microseconds
>>> n_pts = 1000 #interpolated points in solution
>>> sol_dop = rq.solve_time(s, end_time, n_pts, doppler=True)
>>> sol_dop_nosum = rq.solve_time(s, end_time, n_pts, doppler=True, sum_
↪doppler=False)
>>> print(sol_dop.rho.shape)
(11, 1000, 8)
>>> print(sol_dop_nosum.rho.shape)
(561, 11, 1000, 8)
>>> print(sol_dop.axis_labels)
['(0,1)_detuning', 'time', 'density_matrix']
>>> print(sol_dop_nosum.axis_labels)
['doppler_0', '(0,1)_detuning', 'time', 'density_matrix']
```

CHAPTER

NINE

DEVELOPER DOCUMENTATION

These pages contain documentation relevant to the development of rydiqule. If you with to work on the source code of rydiqule, details relating to policies and tools can be found here.

9.1 Unit Tests

Rydiqule comes bundled with a suite of unit tests that confirm basic functionality of the various components and checks for robustness to erroneous or unexpected arguments. We strive to follow the testing methodology and practices employed by numpy. We agree with the stipulation made there, that

> "Long experience has shown that by far the best time to write the tests is before you write or change the code - this is test-driven development"

9.1.1 pytest

Rydiqule takes advantage of the `pytest` testing framework to run unit and integration tests of the code base. The full test suite is run from the project base directory with the command

```
pytest
```

This command will run all tests in the `tests/` subdirectory as well as docstring examples. These tests cover a wide range of functionality as well as a number of representative integrations that demonstrate how the code can be used to generate end results.

Marks

The tests are marked based on the type of test that is being performed, and `pytest` can be told to only run certain tests. For example, this command will only run tests relating to the steady state solving functionality:

```
pytest -m steady_state
```

You can also exclude a specific group of tests. For example, this command will exclude tests marked as slow.

```
pytest -m "not slow"
```

Marks specifications can be combined using standard boolean keywords as well. The following will run all the time tests that are not slow.

```
pytest -m "time and not slow"
```

The available marks can be listed using `pytest --markers`. The markers we use are

Table 9.1: Markers

Marker	Description
slow	Marks a test as taking a long time to run
high_memory	Marks a test needing a lot of RAM
steady_state	Marks a test as using the steady-state solver
time	Marks a test as using the time solver
doppler	Marks a test that incorporates Doppler averaging.
experiments	Marks a test that represents a full experiment.
solution	Marks a test of the `Solution` object and methods.
util	Marks a test of the ancillary utilities.
structure	Marks a test of the definition of the atomic system.
exception	Marks a test of error handling.
backend	Marks a test that uses an optional backend
dev	Used to temporarily mark a single test that is being developed so it can run independently.

Docstring Rounding

Pytest also runs all examples in the docstrings using python's built-n `doctest` module. This module runs the provided code and does simple string matching of the output to confirm results. Given rydiqule is a numerical computation package that deals heavily with floating point numbers, there will be small differences in results based on platform and even exact dependency libraries used. To prevent spurious errors, doctest has been configured to ignore digits of higher precision in the result than what is provided in the source. For example, a computed result of `9.333453` will successfully match against the docstring value of `9.333`.

Coverage

If you install the `pytest-cov` plugin, you can check code coverage of the tests by modifying the command to read.

```
pytest --cov=rydiqule
```

Durations

If you want to see which tests take the longest to complete, you can use the `--durations=n` flag to give the `n` longest time tests:

```
pytest --durations=3
```

Settings the `durations` flag to 0 will cause pytest to report the time taken for all tests run.

9.2 Type Hinting

Rydiqule employs the optional type hinting capabilities of python. These type annotations are not checked or enforced at runtime by python itself. Rather, they provide hints to fellow programmers and users about the types of function arguments, return types, and class variables.

We use the pyright static type checking library to read these hints and catch type errors within the code base. The easiest way to use this checker locally is via the VS Code Pylance extensions, which defaults to the pyright type checker and will automatically run when configuration options are detected in the root `pyproject.toml`.

Note that python is still a dynamic, duck-typed language, and rydiqule employs some features that are perfectly valid python which are not expressible in the type hinting system. In these situations, we err on the side of type hints being primarily documentation, and do our best to not obfuscate functioning code merely to satisfy the type checker.

We also have configurations for using the mypy static type checking library. To run this check locally, install the `mypy` python package and run the following command from the package root directory.

```
mypy
```

This command will automatically read configuration options set in the `pyproject.toml` file. Further optional flags can be passed to the command to override or add optional behaviors. Initial run of the mypy takes some time, however subsequent runs take advantage of local caching to increase analysis speed. Using the mypy daemon mode can further increase analysis speed if necessary.

An html report of the mypy coverage can be generated using the following command.

```
mypy --html-report .mypy_report
```

This command will store the html pages in the specified directory `.mypy_report`. Note that this command takes a long time to run every time, as it cannot use the cache.

9.3 Linting

We use the `ruff` linting package to help enforce code style and consistent readability. It can be run locally from the project root folder by calling the command `ruff check .`. It pulls options for running the command from the `pyproject.toml` configuration file.

If you intend to work on the rydiqule codebase, it is good practice to incorporate automatic linting within your code editor. All code submitted to rydiqule must pass the lint check before acceptance.

Linting can be disabled for a single line by using the `# noqa` tag at the end of the line. Specific error codes can be specified, which should be preferred. For example, `# noqa: E501` ignores only line length errors.

9.4 Building the Documentation

This section describes how to build the documentation locally. A web-hosted copy of the docs is available at https://rydiqule.readthedocs.io/ and should generally be used. These instructions are provided for local testing and development purposes.

The Rydiqule documentation is built locally from the source repository using `sphinx`. To do so, you will need to install the `docs` optional dependencies.

9.4.1 html

An html webpage version of the documentation formatted in the read-the-docs style can be made by running the following command from the `docs/` subdirectory.

```
make html
```

The output will be located in the `docs/build/html/` subdirectory. The home page is `index.html`. The html documentation has the best formatting by default and is the easiest to use.

9.4.2 latexpdf

A pdf version of the documentation can be built using

```
make latexpdf
```

The output will be located in `docs/build/latex` and is called `rydiqule.pdf`. Note that building the pdf requires `perl` and a functioning latex installation with the `latexmk` package. You will also require the GNU FreeFont collection. On Windows, these can be installed manually at the system level or via the MikTeX package `gnu-freefont`. This build also requires a great many other latex packages in addition to `latexmk`. It is easiest to install these packages on the fly as needed, if your latex distribution supports that.

Given the difficulty of building this type of documentation locally, a copy can be downloaded from the documentation website.

9.4.3 latex

It is also possible to build the pdf docs in stages, allowing for more control of the process. This is also how the docs are built on readthedocs, allowing for more accurate reproduction of results there.

First, build the latex for the docs pdf.

```
make latex
```

Then change directory to the `docs/build/latex` directory and run the following `latexmk` command.

```
latexmk -r latexmkrc -pdf -f -dvi- -ps- -jobname=rydiqule -interaction=nonstopmode
```

This workflow largely recreates the `latexpdf` workflow, but invokes options that ensure errors do not stop the build.

9.4.4 epub

There is also the ability to build the documentation in the EPUB format, if desired.

```
make epub
```

This version of the documentation is also available for download on ReadTheDocs.

9.5 Contributing

We actively encourage contributions and collaboration in the development of rydiqule. Unforunately, rydiqule's development is done privately, *for reasons*. If you would like to submit a PR for all but the most trivial of changes, please e-mail us directly (david.h.meyer3.civ@army.mil or kevin.c.cox29.civ@army.mil) so we can discuss the collaboration.

9.5.1 Tips for a successful contribution

If you would like to submit a pull request that improves or fixes rydiqule, first off, thank you! A wider set of contributors to a project significantly improves the quality of the project as well as its pace of development.

Secondly, because of limitations placed on rydiqule's primary developers, development deviates from a typical open-source project in that day-to-day work is done privately, with bulk public code releases. The most important difference for a potential contributor is that early communication with the core developers is even more important so we can advise how to move forward.

Finally, writing code, especially a PR to someone else's project, is similar to writing a journal article for peer-review. The (code) reviewer needs to understand what you have done and why. Ultimately you need to *persuade* them to accept. Below is a general list of tips for producing a PR that can be merged quickly.

Ask *before* doing a lot of work

Unlike paper reviews, the code review process is not blind. As such, and as is common to any open-source project, it is good form to reach out to the developers *before* submitting a PR with significant changes, via an issue on github or a direct message to the devs. This can save you a lot of time as it allows the devs to provide high-level guidance on what approach is likely to work and/or be accepted. It avoids duplicated effort if we are already working on a solution. It avoids wasted effort on things we have already rejected as not viable. It also provides an opportunity for us to directly collaborate and help. Finally, it allows us to set expectations on scope, such as clarifying when a unit test isn't necessary.

Make things digestible

In much the same way a paper with multiple pages of non-stop equations can be hard to follow, submitting a PR with 1000s of changed lines of code, especially without having reached out first, makes the PR very difficult to digest. And we will not accept code we do not understand.

The key to digestible PRs is breaking changes down into small, logical, digestible steps.

- PRs are most like sections of a paper. Ensure that each PR addresses a single task. For example, don't implement unrelated features in a single PR, or undertake large refactors alongside a new feature. It is OK (even expected) to have a PR depend on another to be merged first.
- Commits are most like paragraphs in a paper. You should tend to use many small commits within a PR, ensuring each commit has a single purpose and a sufficiently detailed commit message. For example, moving a file and editing its contents should be separate commits. Updating many inter-related type hints should be a single commit.
- Careful application of git to edit commit history to organize edits after the fact should be considered. After all, it is rare the the order of discovery matches the best order for communicating the result. But note that editing history *after* creating the PR leads to annoyance for others who have already checked out your changes.
- Finally, don't overthink it. If the idea can be communicated in a concise single page, it is generally best to keep it that way instead of fluffing out the manuscript. The same goes for PRs: learn to recognize time-wasting bloat and low-value perfection chasing. If the change is straight-forward, don't waste time checking every box or revising commit messages.

Follow style guides

Submitting a PR that does not follow standard conventions makes it much harder for the reviewer to follow. Even if the underlying work is fundamentally correct, the differences in (ultimately arbitrary) styles consumes the bulk of the review effort.

Please ensure that your code follows the general style guides and practices for rydiqule. This ensures code review focuses on important things, rather than minor details like formatting etc. Aspects include *code linting*, *docstrings and higher-level docs*, and *type hinting*.

Unit tests and examples

Submitting a PR that lacks context or is missing supporting justifications is much akin to submitting an incomplete manuscript. Even if the work represents a significant advancement, the missing details make it hard for the reviewer to understand and therefore accept.

For code, these supporting elements are provided by clear tests and examples, especially in the associated github issue or pull request descriptions. We also expect unit tests and examples to be included into the repository, where appropriate. Please consult *our unit test documentation* to see what conventions we use. Example notebooks should largely follow the style of existing example notebooks in the documentation. Those notebooks live in the /docs/source/examples and /docs/source/intro_nbs directories of the repository.

If you are fixing a bug, unless it is very trivial, we will expect a unit test to accompany the fix that demonstrates the issue and proves the fix works. Ideally, the unit test should be the first commit so it can easily be shown what the issue is.

If you are adding a new feature, appropriate unit testing demonstrating the new feature works will be expected. Depending on the scope of the enhancement, an example added to an existing notebook or more likely a new example notebook will also be expected.

Please be patient

We try very hard to outpace Physical Review timelines. However, like most open-source project developers/maintainers, this is not our primary responsibility. While we strive to at least acknowledge messages quickly, we are busy and it may take time to respond. If things seem stuck, don't be afraid to ping again.

We also strive to be direct in our communication. Especially in code reviews, this can come across as impatient or unappreciative. That is not our intention and we request you not take it personally. We strive to hold code released as a part of rydiqule to a high standard, as it is a general tool with a wide variety of users and use cases. We have to ensure contributions don't have unintended consequences for others, and that they are maintainable *by us* after you have moved on.

Please be prepared for any PR to have many comments, questions, and requested changes before being merged. Niche enhancements that break usability for other applications are unlikely to be merged. While such contributions are important for science, we will likely direct them elsewhere (i.e. a fork, paper supplemental material).

```
%matplotlib inline
```

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

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.transform import Rotation as R
```

```
import rydiqule as rq
from rydiqule.sensor_utils import get_rho_ij
```

CHAPTER

TEN

3-PHOTON RYDBERG EIT

This notebook can be downloaded here.

We demonstrate three photon coherent excitations using the system studied in Taicharoen et. al. PRA 063427 (2019). This is a rubidium vapor with a $5S_{1/2} \rightarrow 5P_{3/2} \rightarrow 5D_{5/2} \rightarrow 28F_{7/2}$ excitation pathway, with corresponding optical fields of 780 nm, 776 nm, and 1260 nm. These fields are labelled probe, dressing, and coupling, respectively.

Here we demonstrate using rydiqule to solve this system under three conditions:

1) Cold atoms
2) Warm atoms, colinear optical beams
3) Warm atoms, doppler-free angles

Because there are three optical fields, the basic coherent feature observed, on resonance, is expected to the absorptive (rather than transmissive like EIT). Going to warm atoms, this feature broadens signficantly, and other coherent features can arise at non-zero detunings of the fields. Going to a Doppler-free excitation in warm atoms, we find that a transmissive feature is observed on resonance. This feature is significantly narrower than those observed in the colinear case.

10.1 Doppler-free, 3 photon excitation

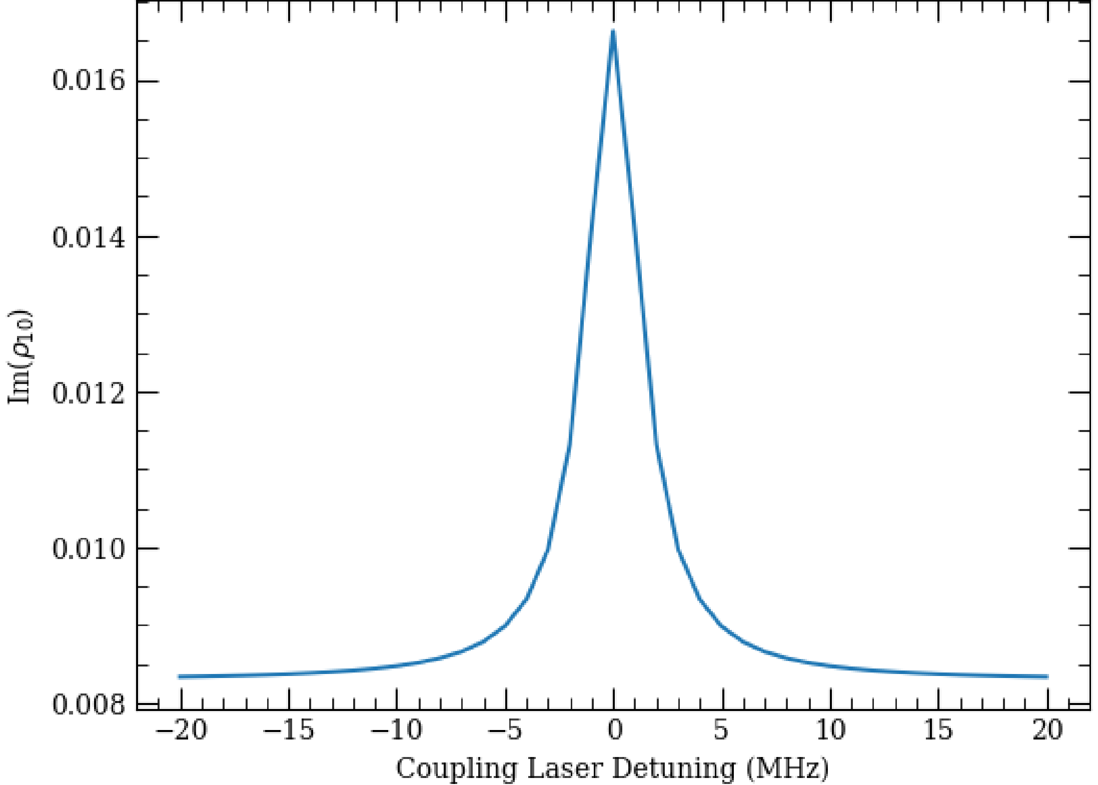
With all three fields resonant, we expect to see Electromagnetically-Induced-Absorption (EIA) instead of EIT.

```
detunings = np.linspace(-20,20,41)

probe = {'states':(0,1), 'rabi_frequency':2*np.pi*0.1,'detuning':2*np.pi*0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*2}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*2, 'detuning':2*np.pi*detunings}

basis_size = 4
gam = np.zeros((basis_size,basis_size),dtype=np.float64)
gam[1,0] = 6
gam[2,1] = 0.66
gam[3,2] = 10e-3
gamma_matrix = 2*np.pi*gam

sensor = rq.Sensor(basis_size)
sensor.add_couplings(probe,couple)
sensor.set_gamma_matrix(gamma_matrix)
```

```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols = rq.solve_steady_state(sensor)
```

```
fig, ax = plt.subplots()
ax.plot(detunings, get_rho_ij(sols.rho,1,0).imag)
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\\rho_{10}$)')
```



10.2 Colinear 3-photon Excitation with Doppler Averaging

We can take our three fields and configure them in the (+,-,-) configuration of Taicharoen (2019). We will need to do Doppler averaging along the colinear axis to get the result. The magnitude of a field's kvector is defined such that it is the magnitude of the Doppler shift associated with the most probable speed of the Maxwell-Boltzmann distribution ($v_P \equiv \sqrt{2k_BT/m}$, where k_B is Boltzmann's constant, T is the gas temperature, and m is the atomic mass). It should have units of Mrad/s like all other specifies quantities.

The following reproduces Figure 2b from Taicharoen et. al. PRA 063427 (2019). Note that having all fields resonant results in an EIA feature, but it now much broader than the Doppler-free case above. If the dressing field is detuning, EIT features are observed.

```
detunings = np.linspace(-200,200,201)

kp = 2*np.pi/780e-3*np.array([1,0,0])
kd = 2*np.pi/776e-3*np.array([-1,0,0])
kc = 2*np.pi/1260e-3*np.array([-1,0,0])
vP = 242.387 # m/s

###
probe = {'states':(0,1), 'rabi_frequency':2*np.pi*10, 'kvec':kp, 'detuning': 0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*25, 'kvec':kd}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*18, 'detuning':2*np.
```

(continues on next page)

(continued from previous page)

```
↪pi*detunings, 'kvec':kc}
###

n = 4
sensor = rq.Sensor(n, vP=vP)
sensor.add_decoherence((1,0), 2*np.pi*6)
sensor.add_decoherence((2,1), 2*np.pi*0.66)
sensor.add_decoherence((3,2), 2*np.pi*10e-3)

sensor.add_couplings(probe,couple)
```

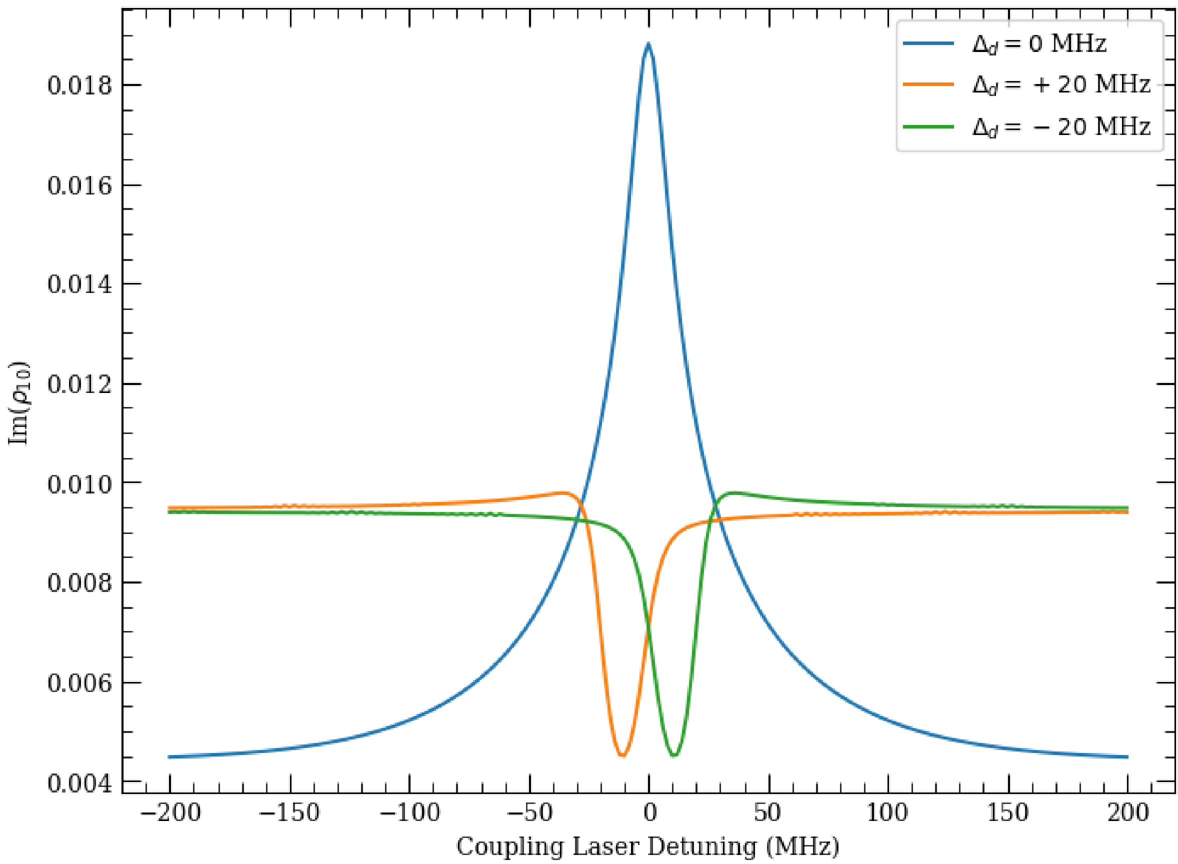
```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols0 = rq.solve_steady_state(sensor,doppler=True)
```

```
dress['detuning'] = 2*np.pi*20
sensor.add_couplings(dress)
solsp20 = rq.solve_steady_state(sensor,doppler=True)
```

```
dress['detuning'] = -2*np.pi*20
sensor.add_couplings(dress)
solsm20 = rq.solve_steady_state(sensor,doppler=True)
```

```
fig, ax = plt.subplots(figsize=(8,6))
ax.plot(detunings, get_rho_ij(sols0.rho,1,0).imag, label="$\\Delta_d= 0$ MHz")
ax.plot(detunings, get_rho_ij(solsp20.rho,1,0).imag, label="$\\Delta_d= +20$ MHz")
ax.plot(detunings, get_rho_ij(solsm20.rho,1,0).imag, label="$\\Delta_d= -20$ MHz")
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
ax.legend()
```

```
<matplotlib.legend.Legend at 0x1e454579690>
```



10.3 Doppler-Free Angles

We can take the same co-propagating system, and instead change the angles of the three beams such that $k_p + k_d + k_c \approx 0$.

We now need to do Doppler averaging in two orthogonal dimensions. Rydiqule automatically detects how many non-zero dimensions are present in the field kvectors and averages over the appropriate number of spatial dimensions.

```
detunings = np.linspace(-20,20,21)

kp = 2*np.pi/780e-3*np.array([1,0,0])
kd = 2*np.pi/776e-3*np.array([-1,0,0])
kc = 2*np.pi/1260e-3*np.array([-1,0,0])
vP = 242.387 # m/s

# rotate to doppler free angles
rd = R.from_euler('z',-35.964,degrees=True)
rc = R.from_euler('z',72.4718,degrees=True)
kdDF = rd.apply(kd)
kcDF = rc.apply(kc)

probe = {'states':(0,1), 'rabi_frequency':2*np.pi*10, 'kvec':kp, 'detuning':0}
dress = {'states':(1,2), 'rabi_frequency':2*np.pi*25, 'kvec':kdDF}
couple = {'states':(2,3), 'rabi_frequency':2*np.pi*18, 'detuning':2*np.
↪pi*detunings, 'kvec':kcDF}

n = 4
sensor = rq.Sensor(n, vP=vP)
```

(continues on next page)

(continued from previous page)

```
sensor.add_decoherence((1,0), 2*np.pi*6)
sensor.add_decoherence((2,1), 2*np.pi*0.66)
sensor.add_decoherence((3,2), 2*np.pi*10e-3)

sensor.add_couplings(probe,couple)
```

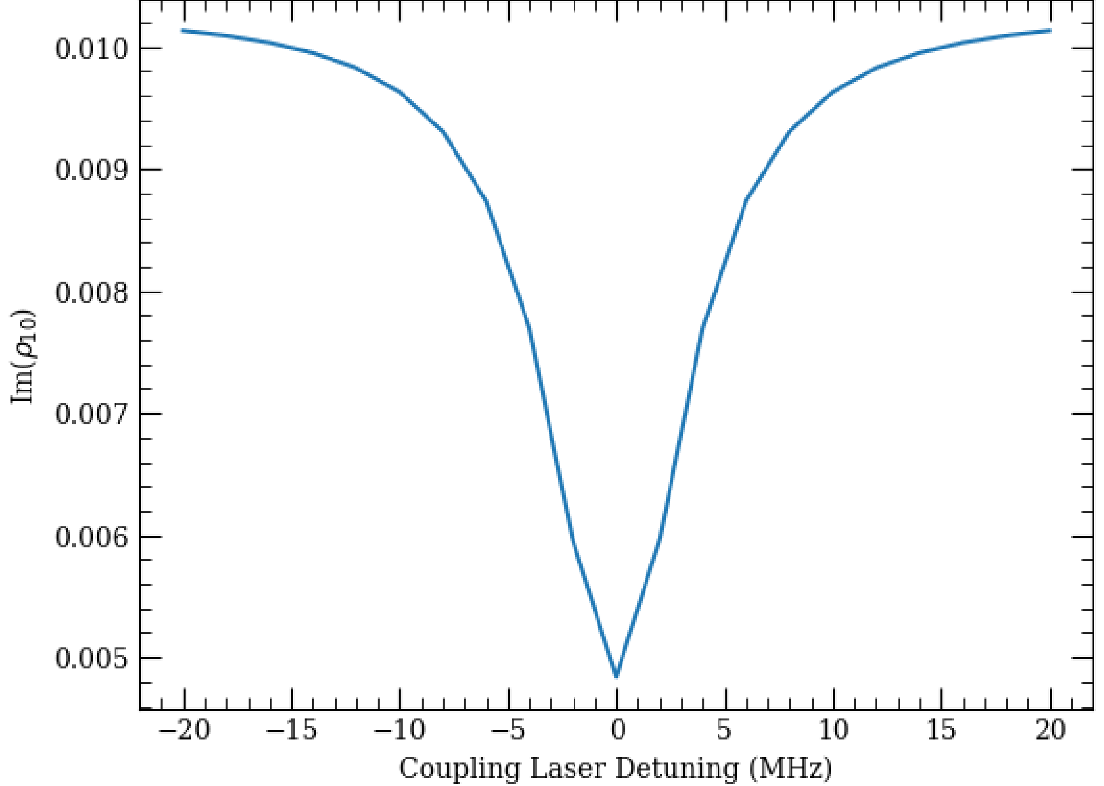
```
print('Residual fractional kvector sum due to round-off errors')
print((kp+kdDF+kcDF)/np.sqrt(kp.dot(kp)))
```

```
Residual fractional kvector sum due to round-off errors
[-1.41309045e-08 -4.99652620e-07  0.00000000e+00]
```

```
dress['detuning'] = 2*np.pi*0
sensor.add_couplings(dress)
sols0DF = rq.solve_steady_state(sensor,doppler=True)
```

```
fig, ax = plt.subplots()
ax.plot(detunings, get_rho_ij(sols0DF.rho,1,0).imag)
ax.set_xlabel("Coupling Laser Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\\rho_{10}$)')
```



We observe a significantly narrower feature than in the collinear case, and it is now EIT instead of EIA.

The authors recognize financial support from the Defense Advanced Research Projects Agency (DARPA). Rydiqule has been approved for unlimited public release by DEVCOM Army Research Laboratory and DARPA. This software is released under the xx licence through the University of Maryland Quantum Technology Center. The views, opinions and/or findings expressed here are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

rydiqule

CHAPTER

ELEVEN

ANALYTIC DOPPLER SOLVER

This notebook demonstrates how to use rydiqule's implementation (`solve_doppler_analytic`) of the analytical doppler solver described in Omar Nagib and Thad G. Walker, *Exact steady state of perturbed open quantum systems*, Phys. Rev. Research 7, 033076 (2025). It contains examples with 1D/2D/3D Doppler averaging along with detailed profiling performance to highlight improvements.

This notebook can be downloaded here.

11.1 1D Doppler Averaging

We will look at a simple case of Autler-Townes splitting in Rubidium-87 with two optical fields (blue and red). Applying an RF tone at different field strengths causes the central peak to split and shift as we scan the probe detuning. We show excellent agreement between numerically integrating the effect of Doppler broadening and the new exact method.

```
%load_ext line_profiler
```

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

```
# parameters for Cell
atom = 'Rb87'

states = [
    rq.ground_state(atom),
    rq.D2_excited(atom),
    rq.A_QState(41,2,5/2),
    rq.A_QState(40,3,7/2)
]


cell = rq.Cell(atom, states)
```

```
# laser parameters
detunings = 2*np.pi*np.linspace(-30,30,201)
Omega_r = 2*np.pi*2
Omega_b = 2*np.pi*5
Omega_rf = 2*np.pi*np.array([0,5,40])

kunit1 = np.array([1,0,0])
kunit2 = np.array([-1,0,0])

red = {'states': (states[0],states[1]), 'detuning': detunings, 'rabi_frequency':␣
```

(continues on next page)

(continued from previous page)

```
↪Omega_r, 'kunit': kunit1}
blue = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency': Omega_b,
↪'kunit': kunit2}
rf = {'states': (states[2],states[3]), 'detuning': 0, 'rabi_frequency': Omega_rf}

cell.add_couplings(red, blue, rf)
```

11.1.1 Method comparison

The analytic averaging method has significantly higher accuracy than the default discrete Riemann sums used by `solve_steady_state`; the default meshing having been chosen to target sufficient accuracy (~1%) for optimal speed. This function compares results from different solve methods to demonstrate the increase in accuracy. We will also time each method for comparison.

```
def compare_accuracy(sol1: np.ndarray, sol2: np.ndarray,
                     rtol: float = 1e-5, atol: float = 1e-7):
    """Helper function for summarizing relative and absolute differences between
    density matrix solutions.

    Note that sol1 is considered the 'correct' solution in the comparison.

    Tolerances are passed to numpy.isclose for defining how close something is."""

    assert sol1.shape == sol2.shape, 'solutions must have same shape to be compared
↪'
    abs_diff = np.abs(sol2 - sol1)
    null_elem = np.isclose(sol1, 0.0)
    sol_ref = sol1.copy()
    sol_ref[null_elem] = 1.0
    rel_diff = abs_diff/np.abs(sol_ref)
    rel_diff[null_elem] = 0.0
    print(f'Abs(diff) max {abs_diff.max():.3e}, mean {abs_diff.mean():.3g}, std
↪{abs_diff.std():.3g}')
    print(f'Rel(diff) max {rel_diff.max():.3e}, mean {rel_diff.mean():.3g}, std
↪{rel_diff.std():.3g}')

    close = np.isclose(sol2, sol1, rtol=rtol, atol=atol)  # element-wise close
    close_sys = close.all(axis=-1)  # density-matrix wise close
    if not close_sys.all():
        print(f'Not close matrix elements {(~close).sum():d} out of {close.size:d}␣
↪total')
        print(f'Not close solutions {(~close_sys).sum():d} out of {close_sys.
↪size:d} total')
        not_close_inds = (~close).nonzero()
        vals, counts = np.unique(not_close_inds[2], return_counts=True)
        for l, v, c in zip(cell.dm_basis(), vals, counts):
            print(f'\tdm element {l:s}-[{v:d}] has {c:d} misses: ' +
                  f'Abs-diff (max, mean, diff) {abs_diff[...,v].max():.3e}, {abs_
↪diff[...,v].mean():.3e}, {abs_diff[...,v].std():.3e}')
        return not_close_inds
```

```
%%time
sol_riemann = rq.solve_steady_state(cell, doppler=True)
```

```
CPU times: total: 2.39 s
Wall time: 991 ms
```

```
dop_mesh_method = {'method': 'split',
                   'width_doppler': 2.0,
                   'n_doppler': 201,
                   'width_coherent': 0.28,
                   'n_coherent': 1001}
```

```
%%time
sol_riemann_finer = rq.solve_steady_state(cell, doppler=True, doppler_mesh_
↪method=dop_mesh_method)
```

```
CPU times: total: 3.17 s
Wall time: 1.86 s
```

```
%%time
sol_exact = rq.solve_doppler_analytic(cell)
```

```
CPU times: total: 46.9 ms
Wall time: 64.1 ms
```

```
bad_inds = compare_accuracy(sol_exact.rho, sol_riemann.rho, rtol=1e-5, atol=7e-5)
```

```
Abs(diff) max 1.797e-04, mean 1.61e-05, std 2.88e-05
Rel(diff) max 1.137e+03, mean 0.784, std 22.4
Not close matrix elements 593 out of 9045 total
Not close solutions 399 out of 603 total
        dm element 10_real-[1] has 220 misses: Abs-diff (max, mean, diff) 1.797e-
↪04, 5.971e-05, 4.712e-05
        dm element 20_real-[2] has 100 misses: Abs-diff (max, mean, diff) 1.615e-
↪04, 3.366e-05, 3.979e-05
        dm element 30_real-[7] has 112 misses: Abs-diff (max, mean, diff) 1.604e-
↪04, 4.080e-05, 3.453e-05
        dm element 10_imag-[9] has 73 misses: Abs-diff (max, mean, diff) 1.082e-04,
↪ 3.516e-05, 2.708e-05
        dm element 11_real-[11] has 49 misses: Abs-diff (max, mean, diff) 1.221e-
↪04, 2.376e-05, 2.820e-05
        dm element 21_real-[14] has 39 misses: Abs-diff (max, mean, diff) 9.864e-
↪05, 2.241e-05, 2.532e-05
```

```
bad_inds_finer = compare_accuracy(sol_exact.rho, sol_riemann_finer.rho, rtol=1e-5,␣
↪atol=7e-5)
```

```
Abs(diff) max 5.529e-06, mean 2.95e-07, std 5.47e-07
Rel(diff) max 3.176e+01, mean 0.0173, std 0.483
```

```
%timeit rq.solve_doppler_analytic(cell)
```

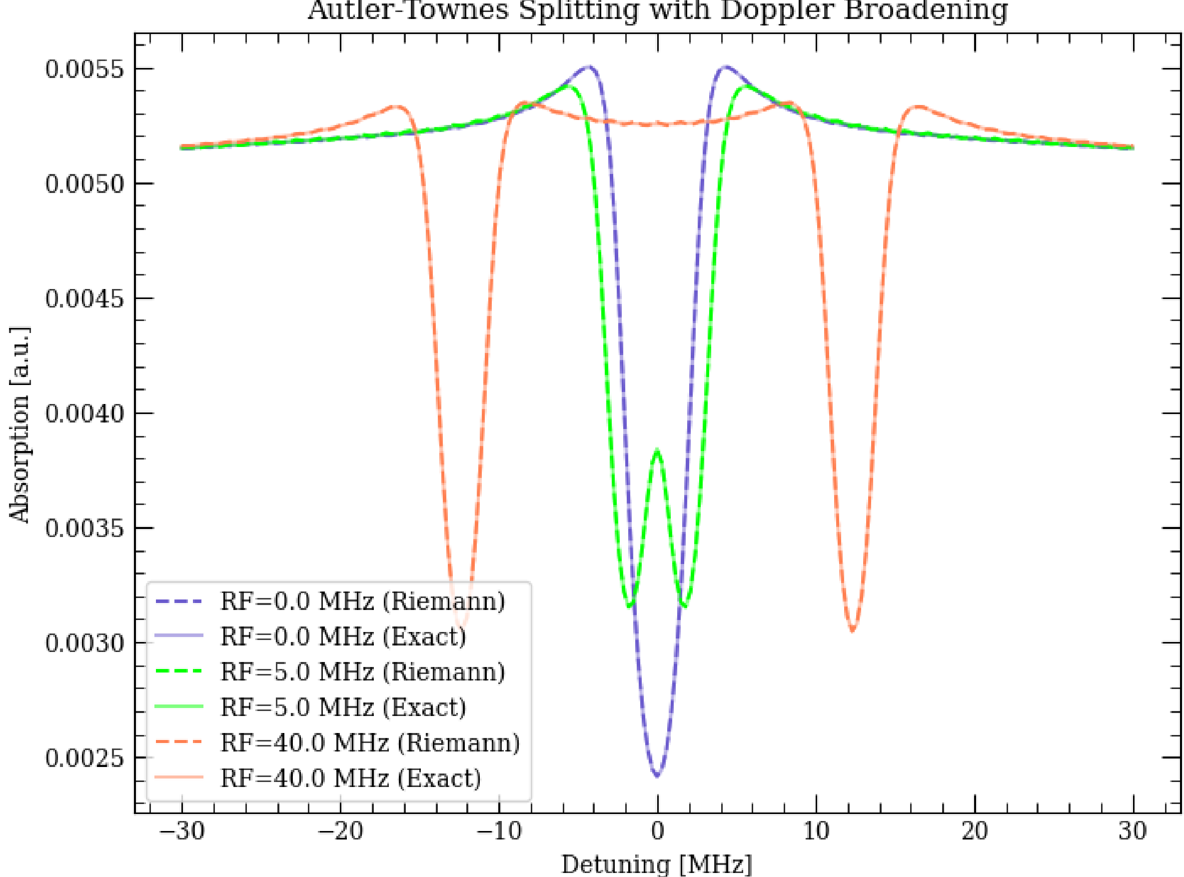
```
55.6 ms ± 2.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
colors=['slateblue', 'lime', 'coral']
plt.figure(figsize=(8,6))
for i in range(len(Omega_rf)):
    plt.plot(detunings/(2*np.pi), sol_riemann.rho_ij(1,0)[:,i].imag,
             label=f'RF={Omega_rf[i]/(2*np.pi):.1f} MHz (Riemann)', c=colors[i],␣
↪linestyle='dashed')
```

(continues on next page)

(continued from previous page)

```
    plt.plot(detunings/(2*np.pi), sol_exact.rho_ij(1,0)[:,i].imag,
             label=f'RF={Omega_rf[i]/(2*np.pi):.1f} MHz (Exact)', c=colors[i],␣
↪alpha=0.5)
plt.xlabel('Detuning [MHz]')
plt.ylabel('Absorption [a.u.]')
plt.legend()
plt.title('Autler-Townes Splitting with Doppler Broadening')
plt.show()
```



11.2 2D Doppler Averaging

This example follows the experimental setup in Glick et al. (https://arxiv.org/pdf/2506.04504). It is a Rb^{85} vapor with a $5S_{1/2} \rightarrow 5P_{1/2} \rightarrow 6S_{1/2} \rightarrow 31P_{1/2}$ excitation pathway and corresponding optical fields of 795 nm, 1324 nm, and 745 nm. These fields are called probe, dressing, and Rydberg, respectively.

We simulate the EIT spectra of the collinear and Doppler-free configurations using the solve_doppler_analytic function that solves the collinear configuration analytically and the Doppler-free configuration using a hybrid analytic/numeric method.

```
# parameters for Cell
atom = 'Rb85'

states = [
    rq.A_QState(5,0,1/2),
    rq.A_QState(5,1,1/2),
    rq.A_QState(6,0,1/2),
```

(continues on next page)

(continued from previous page)

```
    rq.A_QState(31,1,1/2)
]

sensor = rq.Cell(atom, states)
```

```
# laser parameters
detunings = 2*np.pi*np.linspace(-10,10,201)
Omega_p = 2*np.pi*2
Omega_d = 2*np.pi*10
Omega_R = 2*np.pi*1
```

11.2.1 2D Collinear configuration

```
kunit1 = np.array([-1,0,0])
kunit2 = np.array([1,0,0])

probe = {'states': (states[0],states[1]), 'detuning': 0, 'rabi_frequency': Omega_p,
↪ 'kunit': kunit1}
dressing = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d, 'kunit': kunit1}
Rydberg = {'states': (states[2],states[3]), 'detuning': detunings, 'rabi_frequency
↪': Omega_R, 'kunit': kunit2}

sensor.add_couplings(probe, dressing, Rydberg)
sols_col = rq.solve_doppler_analytic(sensor)
```

11.2.2 2D Doppler-free configuration

In this configuration, we show the ability of rydiqule to solve systems using a hybrid analytic/numeric approach, where one user-designated spatial dimension is averaged analytically and the other is averaged numerically.

```
theta = 4.526
phi = 2.556

kunit1 = np.array([-1,0,0])
kunit2 = np.array([-1*np.cos(theta),-1*np.sin(theta),0])
kunit3 = np.array([-1*np.cos(phi),-1*np.sin(phi),0])

probe = {'states': (states[0],states[1]), 'detuning': 0, 'rabi_frequency': Omega_p,
↪ 'kunit': kunit1}
dressing = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d, 'kunit': kunit2}
Rydberg = {'states': (states[2],states[3]), 'detuning': detunings, 'rabi_frequency
↪': Omega_R, 'kunit': kunit3}

sensor.add_couplings(probe, dressing, Rydberg)
```

```
sols_DF = rq.solve_doppler_analytic(sensor, analytic_axis=0)
sols_DF_1 = rq.solve_doppler_analytic(sensor, analytic_axis=1)
```

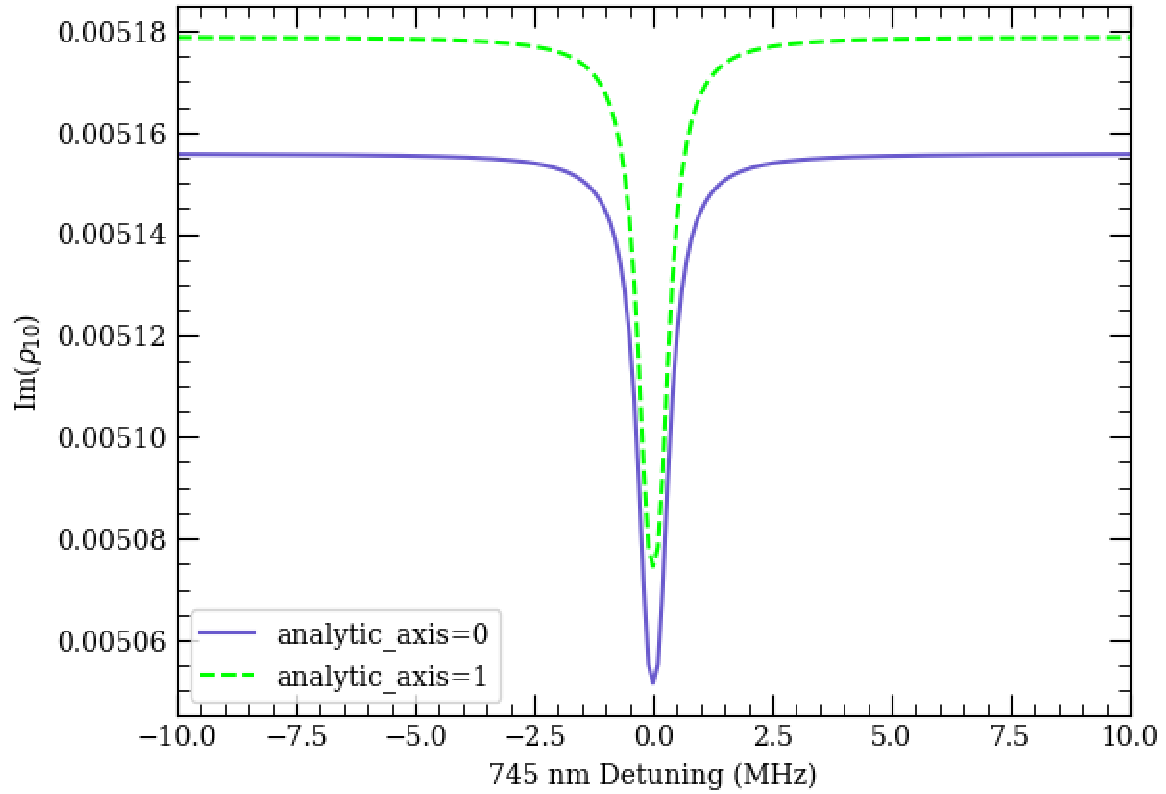
```
fig, ax = plt.subplots()
ax.plot(detunings/(2*np.pi), sols_DF.rho_ij(1,0).imag, c='slateblue', label=
↪"analytic_axis=0")
ax.plot(detunings/(2*np.pi), sols_DF_1.rho_ij(1,0).imag, c='lime', linestyle=
```

(continues on next page)

(continued from previous page)

```
↪'dashed', label="analytic_axis=1")
ax.set_xlabel("745 nm Detuning (MHz)")
ax.set_xlim(-10,10)
ax.set_ylabel(r"Im($\rho_{10}$)")
ax.legend()
```

```
<matplotlib.legend.Legend at 0x23a8b6e0e90>
```

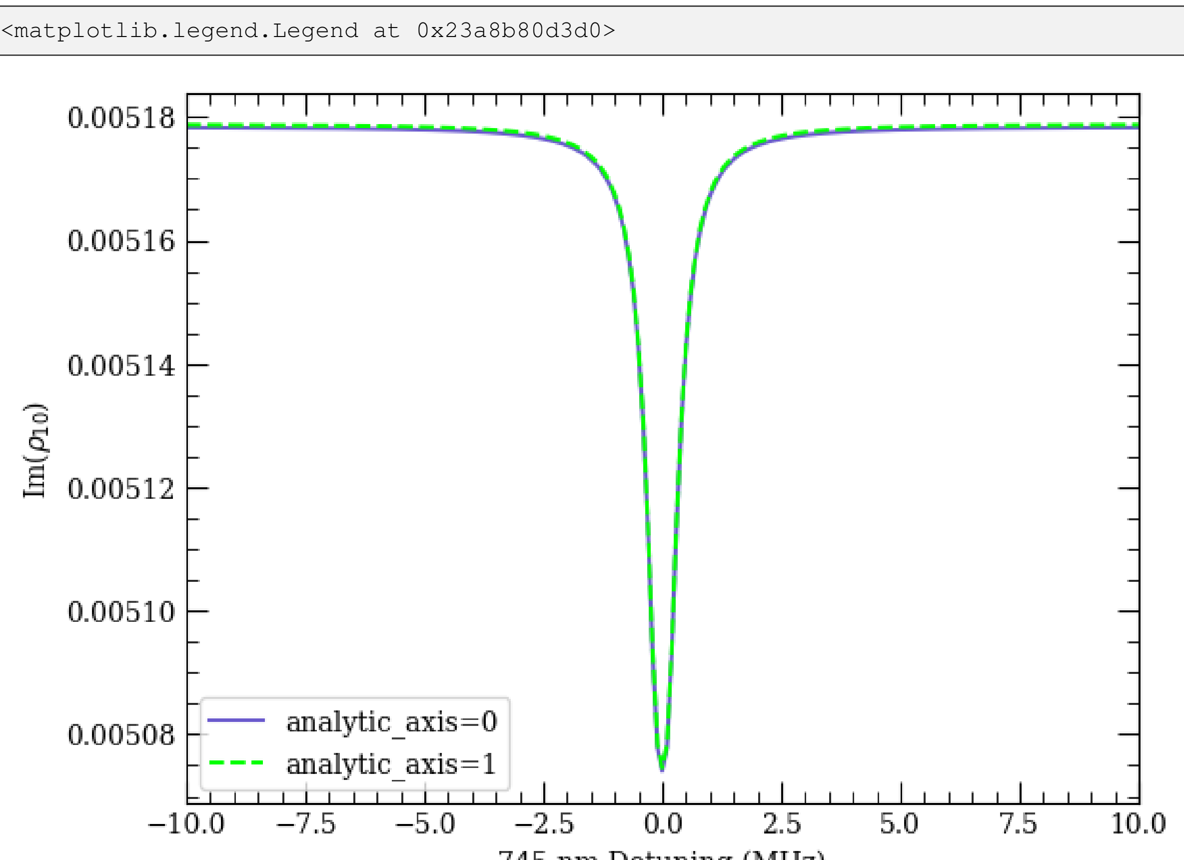


Note that, for some systems, the default mesh is not sufficient to properly capture the tails of the velocity distribution. That is, the distribution normalizes to something slightly less than one. Because the analytic method is much higher accuracy, this can cause a noticeable shift in the solutions depending on which axis is sampled. This can be remedied by increasing the `width_doppler` in the mesh method on the numeric axis.

```
m = {"method":"split", "width_doppler":2.7}
sols_DF_wider = rq.solve_doppler_analytic(sensor, analytic_axis=0, doppler_mesh_
↪method=m)
sols_DF_1_wider = rq.solve_doppler_analytic(sensor, analytic_axis=1, doppler_mesh_
↪method=m)
```

```
fig, ax = plt.subplots()
ax.plot(detunings/(2*np.pi), sols_DF_wider.rho_ij(1,0).imag, c='slateblue', label=
↪"analytic_axis=0")
ax.plot(detunings/(2*np.pi), sols_DF_1_wider.rho_ij(1,0).imag, c='lime', linestyle=
↪'dashed', label="analytic_axis=1")
ax.set_xlabel("745 nm Detuning (MHz)")
ax.set_xlim(-10,10)
ax.set_ylabel(r"Im($\rho_{10}$)")
ax.legend()
```

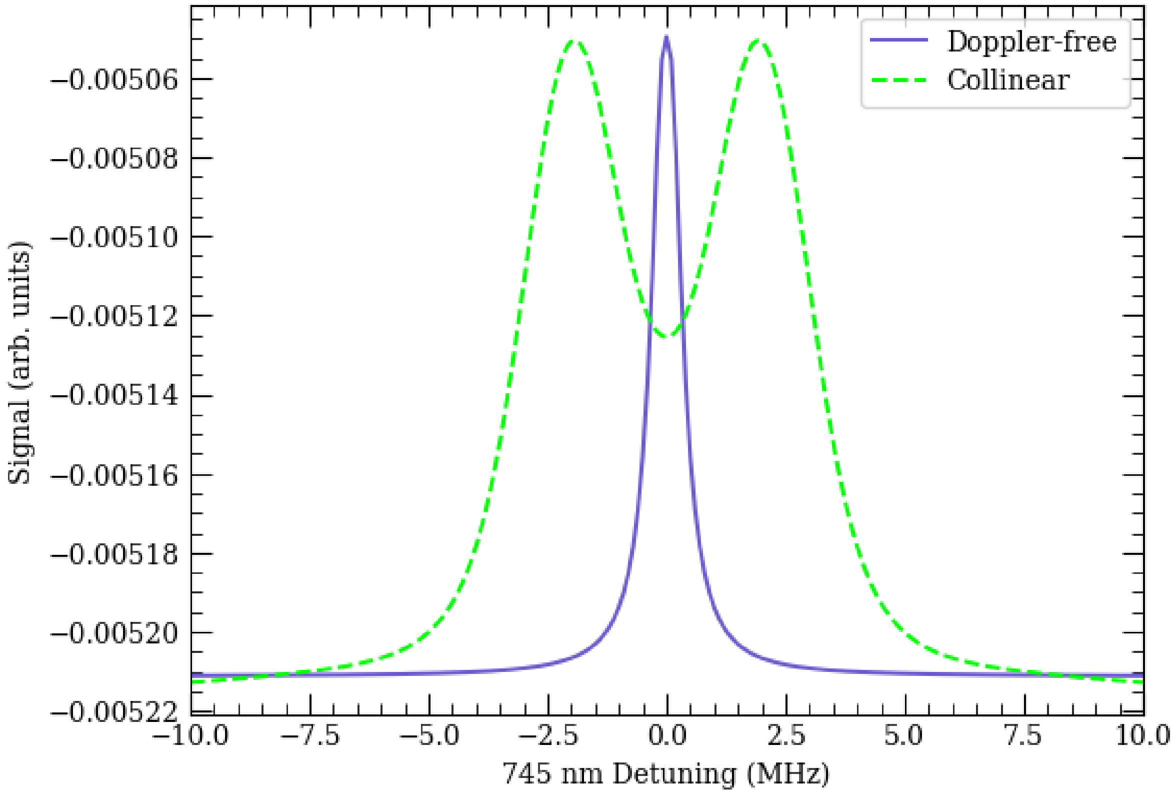
```
<matplotlib.legend.Legend at 0x23a8b80d3d0>
```



We can then recreate Figure 5b in the Glick et al. paper.

```
fig, ax = plt.subplots()
# note that paper figure has y-normalized data for both traces, here we just match␣
↪DF scale/offset to Colinear data
ax.plot(detunings/(2*np.pi), (-1.55*(sols_DF.rho_ij(1,0).imag-5.155e-3)-5.21e-3),␣
↪c='slateblue', label="Doppler-free")
ax.plot(detunings/(2*np.pi), -1*sols_col.rho_ij(1,0).imag, c='lime', linestyle=
↪'dashed', label="Collinear")
ax.set_xlabel("745 nm Detuning (MHz)")
ax.set_xlim(-10,10)
ax.set_ylabel(r"Signal (arb. units)")
ax.legend()
```

```
<matplotlib.legend.Legend at 0x23a8d1c7350>
```



11.2.3 2D computation time and memory footprint improvements

We now show the decrease in both computation time and required memory footprint of the hybrid method in comparison to the existing numeric method.

```
%%time
sols_DF_riemann = rq.solve_steady_state(sensor, doppler=True, doppler_mesh_
↪method=m)
```

```
CPU times: total: 2min 45s
Wall time: 2min 44s
```

```
%%time
sols_DF = rq.solve_doppler_analytic(sensor, analytic_axis=0, doppler_mesh_method =␣
↪m)
```

```
CPU times: total: 6.12 s
Wall time: 4.72 s
```

```
from rydiqule.slicing.slicing import get_slice_num, get_slice_num_hybrid

get_slice_num(n=sensor.basis_size, stack_shape = (201,), doppler_shape=(601, 601),␣
↪sum_doppler=True, weight_doppler=True, debug=True)
```

```
Total available memory: 228 GiB
Min Req memory to solve: 0.68633 GiB
Req memory per EOM: 0.68626 GiB
Req memory for full solve: 137.94 GiB
        Mandatory memory use: 7.0386e-05 GiB
```

(continues on next page)

(continued from previous page)

```
        Memory use for all EOMs: 137.94 GiB
        Full output solution size: 2.2464e-05 GiB
Available memory for sliced solves: 228 GiB
Number of stack slices to be used: 1
```

```
(1, (201, 15))
```

```
get_slice_num_hybrid(n=sensor.basis_size, param_stack_shape=(201,), numeric_
↪doppler_shape=(601,), debug=True)
```

```
--- Analytic Solver Memory Debug ---
Total available RAM: 228 GiB
Min Req memory to solve: 0.01075 GiB
Req memory for full solve: 2.16 GiB
        Full output solution size: 2.3961e-05 GiB
Available memory for sliced calculations: 228 GiB
Calculated minimum slices needed: 1.0
Final number of slices to be used: 1
------------------------------------
```

```
(1, (201, 16))
```

```
print(f'Wall time reduced by x{3*60/5:.0f}')
print(f'Solve memory requirement reduced by x{0.68633/0.01075:.0f}')
```

```
Wall time reduced by x36
Solve memory requirement reduced by x64
```

11.3 3D Doppler Averaging

This example follows the four photon excitation pathway in Cs vapor given in Kondo et al. (https://arxiv.org/pdf/1510.01729). This pathway follows the ladder scheme $6S_{1/2} \rightarrow 6P_{3/2} \rightarrow 7S_{1/2} \rightarrow 8P_{1/2} \rightarrow 52D_{3/2}$ in cesium with corresponding optical fields 852 nm, 1470 nm, 1394 nm, and 1770 nm. These fields are labelled probe, dressing 1, dressing 2, and Rydberg, respectively.

We demonstrate using rydiqule to solve this system with the collinear configuration and with a doppler-free configuration. We include the disclaimer that we ignore the hyperfine structure present in the paper. The purpose of this example is to show that with `solve_doppler_analytic`, doppler averaging over 3 spatial dimensions is now a tractable computation compared to averaging with `solve_steady_state`.

```
# parameters for Cell
atom = 'Cs'

states = [
    rq.A_QState(6,0,1/2),
    rq.A_QState(6,1,3/2),
    rq.A_QState(7,0,1/2),
    rq.A_QState(8,1,1/2),
    rq.A_QState(52,2,3/2)
]

sensor = rq.Cell(atom, states)
```

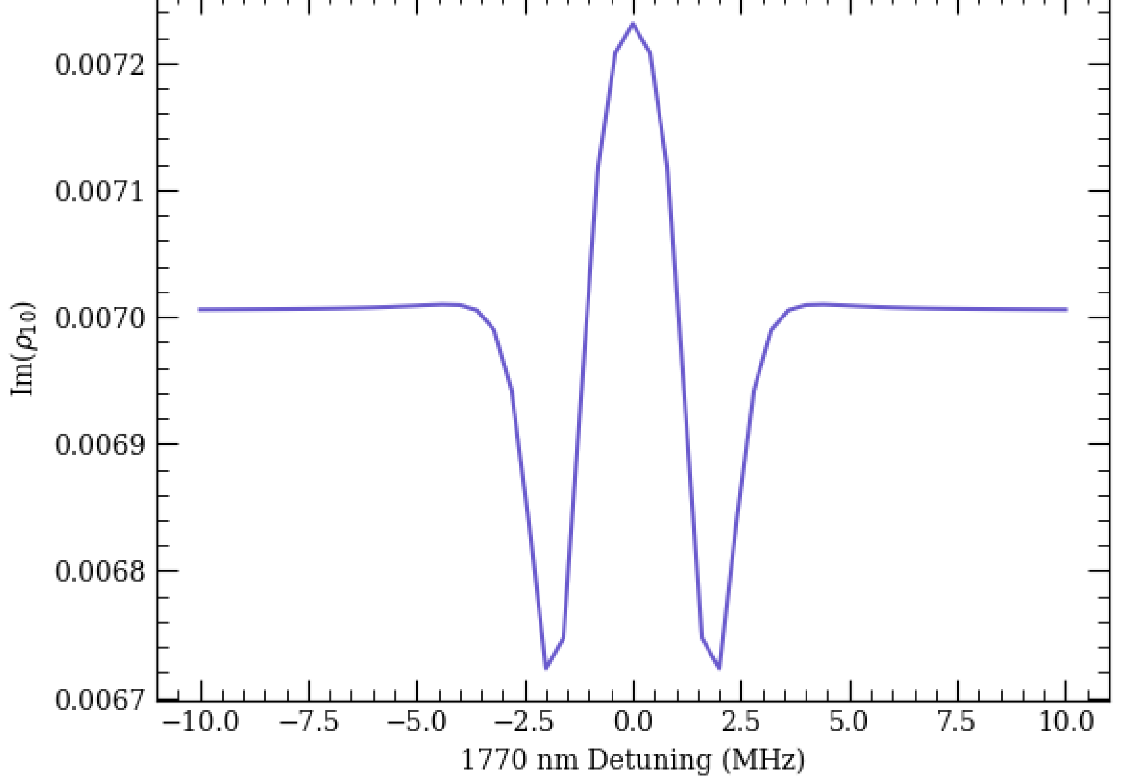
```
# laser parameters
detunings = 2*np.pi*np.linspace(-10,10,51)
Omega_p = 2*np.pi*2
Omega_d1 = 2*np.pi*10
Omega_d2 = 2*np.pi*12
Omega_R = 2*np.pi*1
```

11.3.1 3D Collinear configuration

```
kunit1 = np.array([-1,0,0])
kunit2 = np.array([1,0,0])

probe = {'states': (states[0],states[1]), 'detuning': 0, 'rabi_frequency': Omega_p,
↪ 'kunit': kunit1}
dressing1 = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d1, 'kunit': kunit2}
dressing2 = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d2, 'kunit': kunit2}
Rydberg = {'states': (states[2],states[3]), 'detuning': detunings, 'rabi_frequency
↪': Omega_R, 'kunit': kunit2}

sensor.add_couplings(probe, dressing1, dressing2, Rydberg)
sols_col = rq.solve_doppler_analytic(sensor)
```

```
fig, ax = plt.subplots()
ax.plot(detunings/(2*np.pi), sols_col.rho_ij(1,0).imag, c='slateblue')
ax.set_xlabel("1770 nm Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\\rho_{10}$)')
```



11.3.2 3D Doppler-free configuration

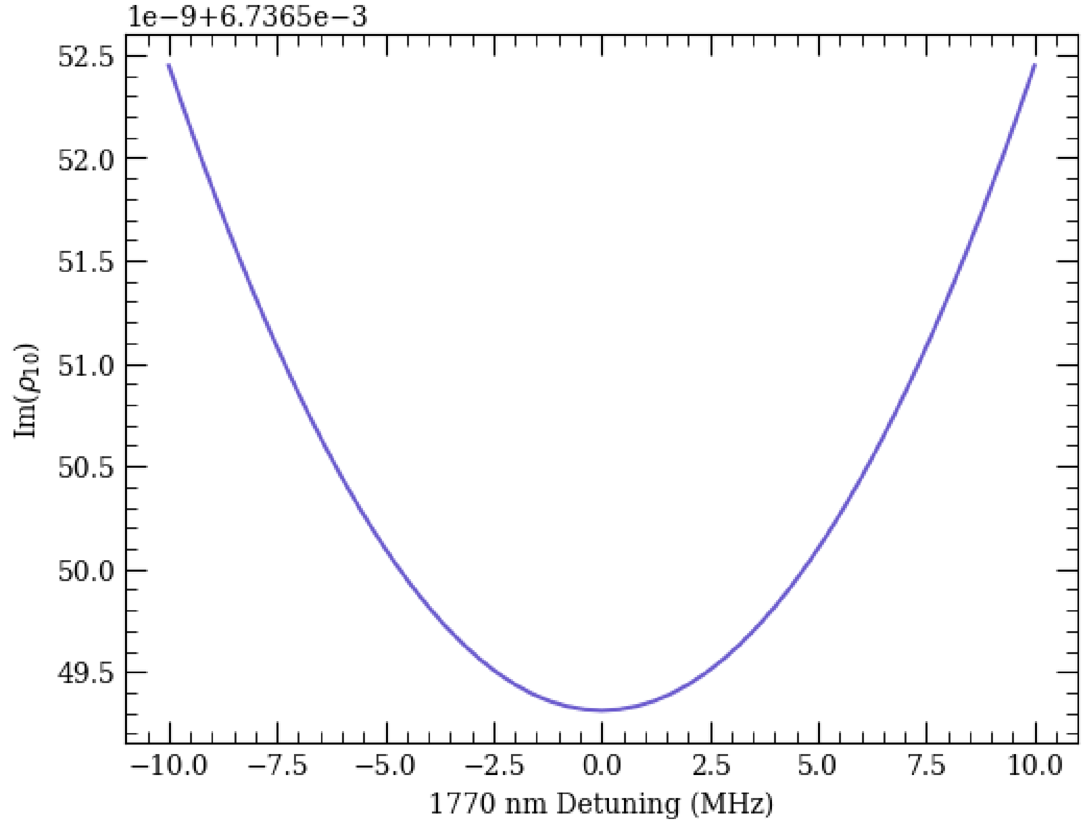
```
# 4 photon
detunings = 2*np.pi*np.linspace(-10,10,51)

theta_p, phi_p = np.pi/2, 0
theta_d1, phi_d1 = 0.3834, np.pi
theta_d2, phi_d2 = 2.1039, 2.6072
theta_R, phi_R = 2.0598, 3.8249

kunitp = np.array([np.sin(theta_p)*np.cos(phi_p), np.sin(theta_p)*np.sin(phi_p),␣
↪np.cos(theta_p)])
kunitd1 = np.array([np.sin(theta_d1)*np.cos(phi_d1), np.sin(theta_d1)*np.sin(phi_
↪d1), np.cos(theta_d1)])
kunitd2 = np.array([np.sin(theta_d2)*np.cos(phi_d2), np.sin(theta_d2)*np.sin(phi_
↪d2), np.cos(theta_d2)])
kunitR = np.array([np.sin(theta_R)*np.cos(phi_R), np.sin(theta_R)*np.sin(phi_R),␣
↪np.cos(theta_R)])

probe = {'states': (states[0],states[1]), 'detuning': 0, 'rabi_frequency': Omega_p,
↪ 'kunit': kunitp}
dressing1 = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d1, 'kunit': kunitd1}
dressing2 = {'states': (states[1],states[2]), 'detuning': 0, 'rabi_frequency':␣
↪Omega_d2, 'kunit': kunitd2}
Rydberg = {'states': (states[2],states[3]), 'detuning': detunings, 'rabi_frequency
↪': Omega_R, 'kunit': kunitR}

sensor.add_couplings(probe, dressing1, dressing2, Rydberg)
```

```
%%time
sols_DF = rq.solve_doppler_analytic(sensor, analytic_axis=0)
```

```
Breaking parameter stack into 4 slices...
CPU times: total: 24min 40s
Wall time: 24min 36s
```

```
fig, ax = plt.subplots()
ax.plot(detunings/(2*np.pi), sols_DF.rho_ij(1,0).imag, c='slateblue')
ax.set_xlabel("1770 nm Detuning (MHz)")
ax.set_ylabel(r"Im($\rho_{10}$)")
```

```
Text(0, 0.5, 'Im($\\rho_{10}$)')
```



11.3.3 3D memory footprint comparison

Here we only directly compare the memory footprint of the two methods, since the discrete sampling method results in too large a system to solve on even a moderately-sized workstation.

```
from rydiqule.slicing.slicing import get_slice_num, get_slice_num_hybrid

get_slice_num(n=sensor.basis_size, stack_shape = (51,), doppler_shape=(561, 561,␣
↪561), sum_doppler=True, weight_doppler=True, debug=True)
```

```
Total available memory: 231.27 GiB
Min Req memory to solve: 820.85 GiB
Req memory per EOM: 820.85 GiB
Req memory for full solve: 41863 GiB
        Mandatory memory use: 2.8118e-05 GiB
        Memory use for all EOMs: 41863 GiB
        Full output solution size: 9.1195e-06 GiB
Available memory for sliced solves: 231.27 GiB
```

```
---------------------------------------------------------------------------
RydiqueleError                            Traceback (most recent call last)
Cell In[35], line 3
      1 from rydiqule.slicing.slicing import get_slice_num, get_slice_num_hybrid
----> 3 get_slice_num(n=sensor.basis_size, stack_shape = (51,), doppler_shape=(561,
↪ 561, 561), sum_doppler=True, weight_doppler=True, debug=True)

RydiquleError: System is too large to solve. Need at least 820.8490894585848 GiB
```

```
get_slice_num_hybrid(n=sensor.basis_size, param_stack_shape=(51,), numeric_doppler_
↪shape=(561,561), debug=True)
```

```
--- Analytic Solver Memory Debug ---
Total available RAM: 231.2 GiB
Min Req memory to solve: 13.54 GiB
Req memory for full solve: 690.6 GiB
        Full output solution size: 9.4995e-06 GiB
Available memory for sliced calculations: 231.2 GiB
Calculated minimum slices needed: 3.0
Final number of slices to be used: 3
------------------------------------
```

```
(3, (51, 25))
```

```
print(f'Solve memory requirement reduced by x{820.85/13.54:.0f}')
```

```
Solve memory requirement reduced by x61
```

```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.1.1.dev18
Installation Path:    ~\src\rydiqule_public\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.2.5
SciPy Version:        1.16.0
Matplotlib Version:   3.10.0
ARC Version:          3.6.0
Python Version:       3.11.10
Python Install Path:  ~\miniconda3\envs\rq_public
Platform Info:        Windows (AMD64)
CPU Count and Freq:   16 @ 3.91 GHz
Total System Memory:  256 GB
```

CHAPTER

TWELVE

CALCULATING SNR

This notebook can be downloaded here.

Rydiqule contains the function `rydiqule.get_snr()` function that will take a Sensor or Cell and calculate the expected SNR for one axis of the solve. Below we demonstrate the use of this function to numerically confirm the analytic results of Meyer et. al. PRA 104, 043103 (2021) Eqs. 12 & 13:

$$\Omega_p^{(\text{opt})} \approx \sqrt{\Gamma(2\gamma + \Gamma_r + \Gamma_c)}$$

$$\Omega_c^{(\text{opt})} \approx \sqrt{2}\Omega_p^{(\text{opt})}$$

These show the optical probe and coupling Rabi frequencies for resolving Rydberg state shifts in a 2-color Rydberg EIT measurement, in an optically-thin with no Doppler broadening.

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

Manually define representative kappa and eta constants for a Rb85 sensor. These are necessary to find the SNR in experimental units and must be supplied by the user when calculating using a Sensor. If using a Cell, these constants are automatically calculated and do not need to be passed to `get_snr`.

The definition of these numerical factors is found in Meyer et. al. PRA 104, 043103 (2021) Eqs. 5 & 7.

$$\kappa = \frac{\omega_p n \mu^2}{2c\epsilon_0\hbar}$$

$$\eta = \sqrt{\frac{\omega\mu^2}{2c\epsilon_0\hbar A}}$$

```
kappa = 28974.8787
eta = 0.00135882
probe_freq = 2.416e9
```

12.1 1D Optimum

Here we demonstrate calculating the SNR for resolving a phase shift due to an RF Rydberg coupling vs probe Rabi frequency. We have chosen a far-detuned RF coupling to ensure Stark shifts are linear.

```
##Set up a simplified Rb Sensor
basis_size = 4
Rb_sensor = rq.Sensor(basis_size)
Rb_sensor.set_experiment_values(probe_freq = probe_freq,
                              kappa = kappa, eta = eta, cell_length = .000001)
```

(continues on next page)

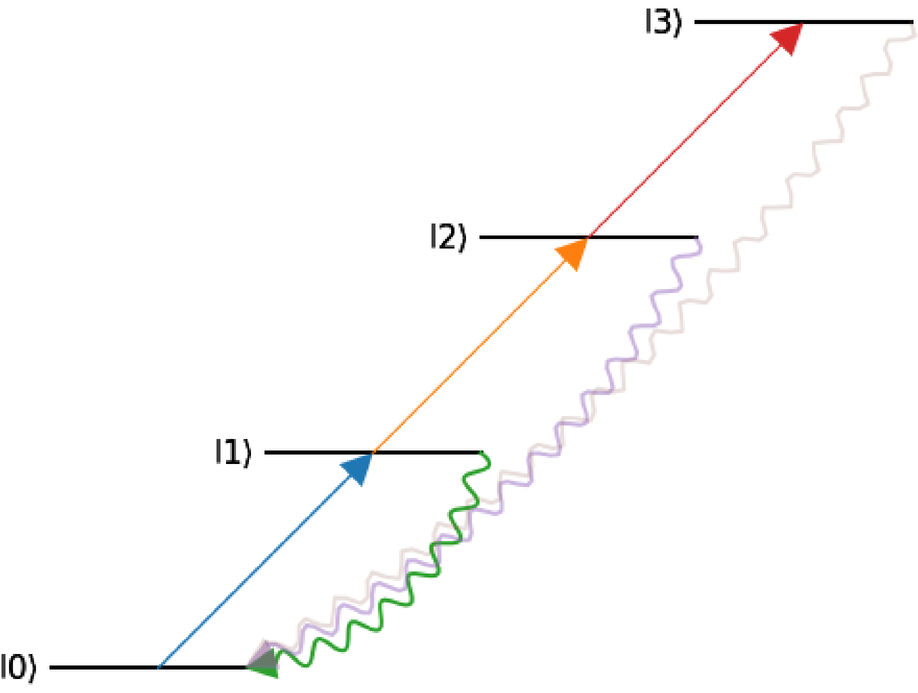
(continued from previous page)

```
red_rabi = np.linspace(0.1,6,100)
blue_rabi = np.linspace(0.1,4,101)
blue_rabi_1 = 1
my_step = np.array([1, 1.1])
probe = {'states': (0,1), 'rabi_frequency': red_rabi, 'detuning': 0, 'label':
↪'probe'}
couple = {'states': (1,2), 'rabi_frequency':blue_rabi_1, 'detuning': 0, 'label':
↪'couple'}
rf = {'states': (2,3), 'rabi_frequency': my_step, 'detuning':20, 'label': 'rf'}

Rb_sensor.add_couplings(probe,couple, rf)

#simplify the gamma matrix to match predictions
gam = np.zeros((basis_size, basis_size))
gam[2,0] = 0.1
gam[3,0] = 0.01
gam[1,0] = 6.0
Rb_sensor.set_gamma_matrix(gam)
```

To calculate the SNR vs a specific parameter, that parameter must be list-like with at least two elements. So calculated vs RF Rabi frequency, we have specified two Rabi frequency values very close to each other to measure the local linear sensitivity. More values can be added to this list to see if sensitivity changes for larger changes in the parameter, which indicates nonlinear response.

```
rq.draw_diagram(Rb_sensor)
```

```
<leveldiagram.ld.LD at 0x2eaf6cd70d0>
```



We call `get_snr` with the Sensor to calculate with, the label of the swept parameter to calculate SNR against, the tuple of the probing transition to get measurable parameters from, which quadrature the probing field is being detected in, and the kappa and eta numerical factors.

```
snrs, param_mesh = rq.get_snr(Rb_sensor, param_label = 'rf_rabi_frequency',
                              phase_quadrature = True)
```

Using `Sensor.axis_labels()` we can identify which axis rydiqule has used for the swept parameters. This allows us to correctly index out the appropriate solutions for analysis. In particular, we need to index the sensitivity axis to get the sensitivity at the second RF Rabi frequency in the list (relative to the first).

```
#print the axis labels
Rb_sensor.axis_labels()
```

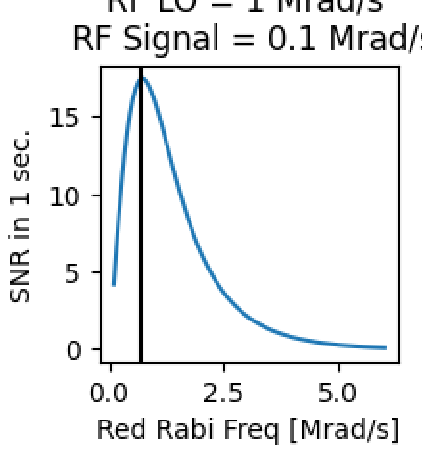
```
['probe_rabi_frequency', 'rf_rabi_frequency']
```

```
snrs_final = snrs[:,1]
param_mesh_final=np.array(param_mesh)[:,:,1]
```

We can plot the SNR as a function of probe rabi frequency. The vertical line represents the analytic optimum value for the probe Rabi frequency.

```
fix, ax = plt.subplots(figsize = (2,2))
ax.plot(param_mesh_final[0], snrs_final)
ax.set_xlabel("Red Rabi Freq [Mrad/s]")
ax.set_ylabel("SNR in 1 sec.")
ax.set_title('RF LO = 1 Mrad/s \n RF Signal = 0.1 Mrad/s')
ax.axvline(blue_rabi_1/np.sqrt(2),0,3, color = 'k')
```

```
<matplotlib.lines.Line2D at 0x2eaf6c6fee0>
```



12.2 2D Optimum - Fnd Optimized Ω_p and Ω_c for best SNR

We can also calculate the SNR versus many different axis. Here we calculate versus both the probe and coupling Rabi frequencies.

```
couple = {'states': (1,2), 'rabi_frequency': blue_rabi, 'detuning': 0, 'label':
↪'couple'}

Rb_sensor.add_couplings(probe,couple, rf)
```

```
snrs, param_mesh = rq.get_snr(Rb_sensor, param_label = 'rf_rabi_frequency', phase_
↪quadrature = True)
```

```
Rb_sensor.axis_labels()
```

```
['probe_rabi_frequency', 'couple_rabi_frequency', 'rf_rabi_frequency']
```

```
snrs_final = snrs[:,:,1]
param_mesh_final=np.array(param_mesh)[:,:,:,1]
```

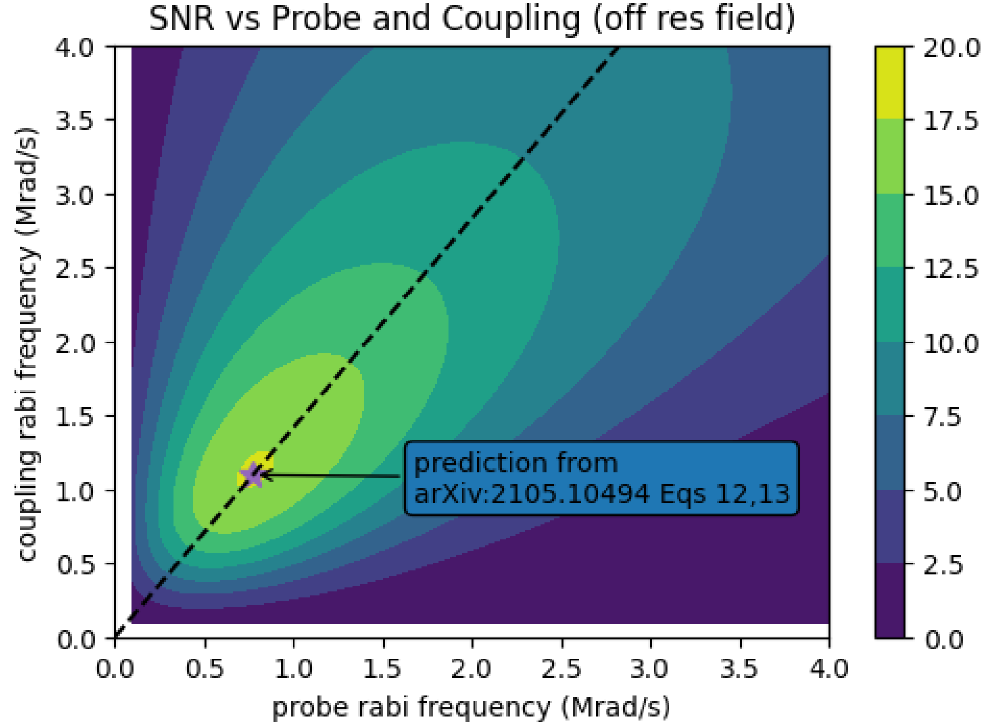
```
predictedOptimumProbe = np.sqrt(gam[1,0]*gam[2,0])
predictedOptimumCouple = np.sqrt(2*gam[1,0]*gam[2,0])
print(f'Predicted optimum probe Rabi frequency: {predictedOptimumProbe:.3f} Mrad/s
↪')
print(f'Predicted optimum coupling Rabi frequency: {predictedOptimumCouple:.3f}␣
↪Mrad/s')
```

```
Predicted optimum probe Rabi frequency: 0.775 Mrad/s
Predicted optimum coupling Rabi frequency: 1.095 Mrad/s
```

We plot the SNR versus both Rabi frequencies using a contour plot. We have overlaid the analytic predictions for the optimal SNR. Compare Figure 5(a) of Meyer et. al.

```
fig, ax = plt.subplots(figsize = (6,4))
CS = ax.contourf(param_mesh_final[0], param_mesh_final[1], snrs_final)
fig.colorbar(CS)
ax.set_xlabel('probe rabi frequency (Mrad/s)')
ax.set_ylabel('coupling rabi frequency (Mrad/s)')
ax.plot(predictedOptimumProbe, predictedOptimumCouple, '*', color = 'C4',␣
↪markersize = 10)
ax.plot([0,10,20], [0,np.sqrt(2)*10,np.sqrt(2)*20 ],'--', color = 'black')
ax.set_title("SNR vs Probe and Coupling (off res field)")
ax.set_ylim((0,4))
ax.set_xlim((0,4))

ax.annotate("prediction from\narXiv:2105.10494 Eqs 12,13",
            xy=(predictedOptimumProbe, predictedOptimumCouple), xycoords='data',
            xytext=(60,-10), textcoords='offset points',
            arrowprops=dict(arrowstyle='->'),
             bbox=dict(boxstyle='round')
            )
```

```
Text(60, -10, 'prediction from\narXiv:2105.10494 Eqs 12,13')
```




```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.2.0.dev49+g12b02b0ad.d20260120
Installation Path:    ~\src\rydiqule_public\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.2.6
SciPy Version:        1.15.3
Matplotlib Version:   3.10.8
ARC Version:          3.9.0
Python Version:       3.10.19
Python Install Path:  ~\src\rydiqule_public\.venv\Scripts
Platform Info:        Windows (AMD64)
CPU Count and Freq:   16 @ 3.91 GHz
Total System Memory:  256 GB
```

CHAPTER THIRTEEN

SIMPLE NMOR EXAMPLES

The goal of this notebook is to show how Rydiqule can be used to define and solve atomic systems that exhibit Nonlinear Magneto-Optical Rotation (NMOR). These systems generally involve a single optical field coupling two hyperfine states with an applied axial magnetic field that breaks the degeneracy of the magnetic sublevels.

As such, an accurate model includes all sublevels of each transition, with the optical coupling being applied to the sub-levels according to the appropriate Clebsch-Gordon coefficients. This is accomplished by using rydiqule's coupling groups functionality with an appropriate clebsch-gordon coefficient dictionary mapping.

One must also be able to apply Larmor shifts to the magnetic sublevels due to the interaction with the magnetic field. This is accomplished by applying energy shifts to each sublevel of a manifold, using a similar dictionary mapping for the energy shift of each state.

The below examples all use the `Sensor` class. This class is largely un-aware of atomic physics, so it is up to the user to specify the correct couplings and interactions when modeling a system.

This notebook can be downloaded here.

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

```
%load_ext line_profiler
```

```
import numpy as np
import itertools
import rydiqule as rq
from arc import CG
import matplotlib.pyplot as plt
```

13.1 A F=0 to F'=1 spectroscopy experiment with $\sigma^{\pm}$ probing light

This example assumes a fixed axial magnetic field and shows the spectroscopic response versus probe detuning.

```
g = ('g', 0)
e = ('e', [-1,0,1])
```

```
s = rq.Sensor([g, e])
```

```
dets = np.linspace(-10,10,21)
rabi_freq = 1
cg = {(g, ('e',-1)):1,
      (g, ('e', 0)):0,
      (g, ('e', 1)):-1}

s.add_coupling((g, e), rabi_frequency=rabi_freq, detuning=2*np.pi*dets, coupling_
↪coefficients=cg, label='probe')
```

```
s.add_decoherence((e, g), 2*np.pi*6.0666, label='F1_gamma')
```

```
larmor_freq = 3
larmor_shifts = {('e', -1): 2*np.pi*larmor_freq,
                 ('e', 1): -2*np.pi*larmor_freq}

s.add_energy_shifts(larmor_shifts)
```

Note that `Sensor.add_coupling` has automatically zipped together all of the individual couplings detunings' for the detuning sweep. So there is only one parameter dimension in the generated hamiltonians (instead of two).

```
hams = s.get_hamiltonian()
print(hams.shape)
```

```
(21, 4, 4)
```

```
solF0F1sig = rq.solve_steady_state(s)
```

Full characterization of an optical field requires four parameters. We prefer the angle-ellipticity parameterization where the field is described by its amplitude, phase, linear polarization angle, and ellipticity. Rydiqule's built-in `coupling_coefficient_observable` uses the defined coupling-coefficients for the desired transition to define the susceptibility of the ensemble that is co-polarized with the input light field. This susceptibility then directly gives the field fractional absorption as well as the phase shift.

To obtain the polarization rotation and ellipticity changes, one must calculate the susceptibility for the orthogonal ensemble polarizability. This is done by providing the coupling-coefficient matrix that corresponds to the orthogonally polarized light field. In this example, this matrix happens to be the absolute value of the co-polarized matrix.

```
aligned_obs = solF0F1sig.coupling_coefficient_matrix((g, ('e', 'all')))
print(aligned_obs)
perp_obs = np.abs(aligned_obs)
print(perp_obs)
```

```
[[ 0.  0.  0.  0.]
 [ 1.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [-1.  0.  0.  0.]]
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 0. 0. 0.]]
```

```
susc = solF0F1sig.coupling_coefficient_observable((g, ('e', 'all')))
susc_perp = solF0F1sig.get_observable(perp_obs)
```

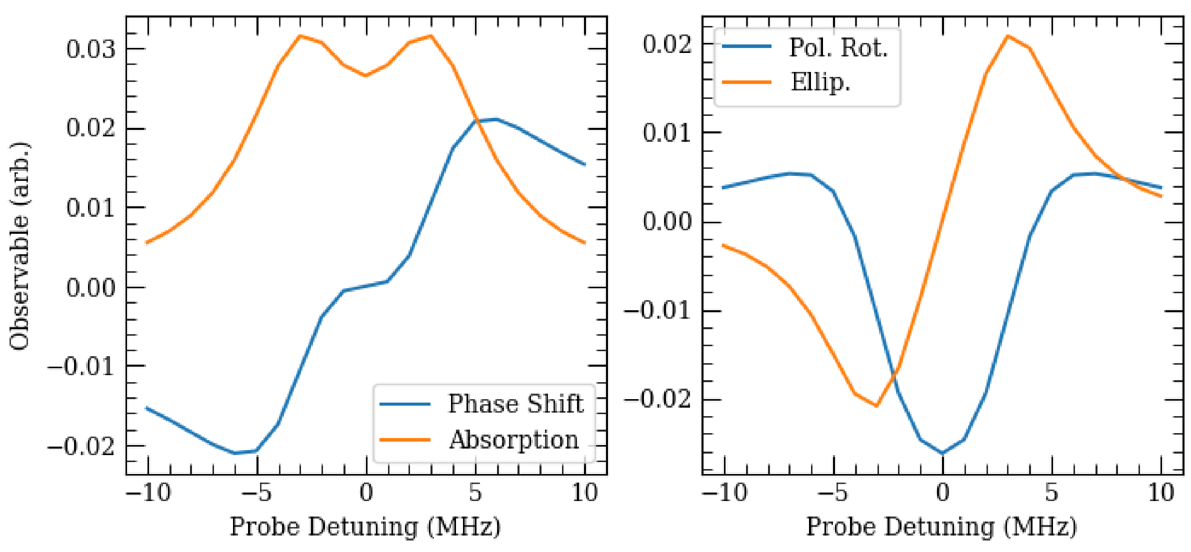
```
fig, (ax, ax2) = plt.subplots(1, 2, figsize=(8, 3.5))

ax.plot(dets, susc.real, label='Phase Shift')
ax.plot(dets, susc.imag, label='Absorption')
ax.legend()
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('Observable (arb.)')

ax2.plot(dets, susc_perp.real, label='Pol. Rot.')
ax2.plot(dets, susc_perp.imag, label='Ellip.')
```

(continues on next page)

(continued from previous page)

```
ax2.legend()
ax2.set_xlabel('Probe Detuning (MHz)')
```

```
Text(0.5, 0, 'Probe Detuning (MHz)')
```



13.2 A F=0 to F'=1 spectroscopy experiment with π polarized probe light

This example is nearly identical to the first example, but the light is now π polarized (meaning the quantization axis has changed to be aligned with the optical polarization). Here the observable corresponds to the phase shift on the light.

```
g = ('g', 0)
e = ('e', [-1,0,1])
```

```
s = rq.Sensor([g,e])
```

```
dets = np.linspace(-10,10,21)
rabi_freq = 1
cg = {(g, ('e',-1)):0,
      (g, ('e', 0)):1,
      (g, ('e', 1)):0}

s.add_coupling((g,e), rabi_frequency=2*np.pi*rabi_freq, detuning=2*np.pi*dets,␣
↪coupling_coefficients=cg, label='probe')
```

```
s.add_decoherence((e,g), 2*np.pi*6.0666, label='F1_gamma')
```

```
solF0F1pi = rq.solve_steady_state(s)
```

```
susc = solF0F1pi.coupling_coefficient_observable((g, ('e', 'all')))
```

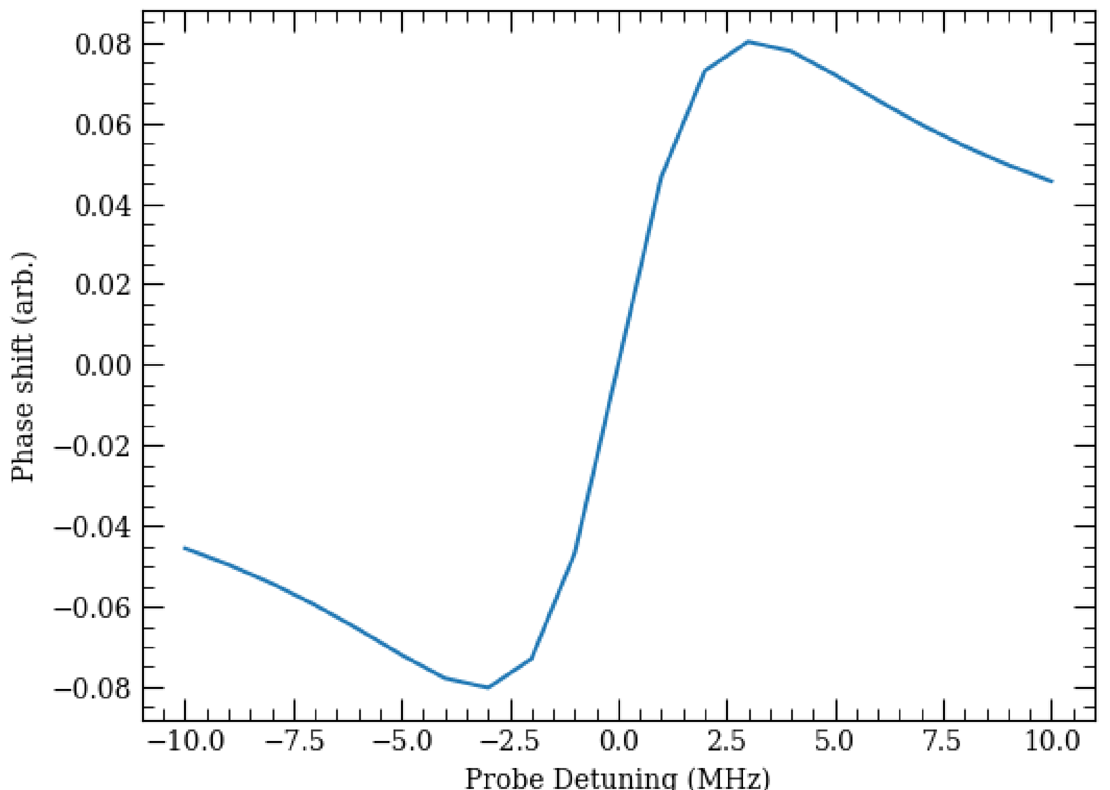
```
fig, ax = plt.subplots(1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(dets, susc.real)
ax.set_xlabel("Probe Detuning (MHz)")
ax.set_ylabel("Phase shift (arb.)")
```

```
Text(0, 0.5, 'Phase shift (arb.)')
```



13.3 A F=1 to F'=0 spectroscopy experiment with $\sigma^{\pm}$ and a static axial magnetic field

This is much like the first example, but the degeneracy of the hyperfine manifolds is reversed between the ground and excited states. In order for this system to produce an interesting result, we add transit broadening. This prevents all the population from being optically pumped into the $F = 1, m_F = 0$ ground state.

```
g = (1, [-1,0,1])
e = (0, 0)
```

```
s = rq.Sensor([g,e])
```

```
dets = np.linspace(-10,10,21)
rabi_freq = 1
cg = {((1,-1), e):1,
      ((1,0), e):0,
      ((1,1), e):-1}

s.add_coupling((g,e), rabi_frequency=2*np.pi*rabi_freq, detuning=2*np.pi*dets,␣
↪coupling_coefficients=cg, label='probe')
```

```
cgg = {(e, (1,-1)): 1/3,
       (e, (1,0)): 1/3,
       (e, (1,1)): 1/3}
s.add_decoherence((e, g), 2*np.pi*6.0666, coupling_coefficients=cgg, label='F0_
↪gamma')

s.add_transit_broadening(2*np.pi*0.1, repop={state:1/3 for state in s.states_with_
↪spec(g)})
```

```
larmor_freq = 3
prefactors = {(1,-1): 2*np.pi,
              (1,1): -2*np.pi}

s.add_energy_shift(g, larmor_freq, prefactors=prefactors)
```

```
solF1F0sig = rq.solve_steady_state(s)
```

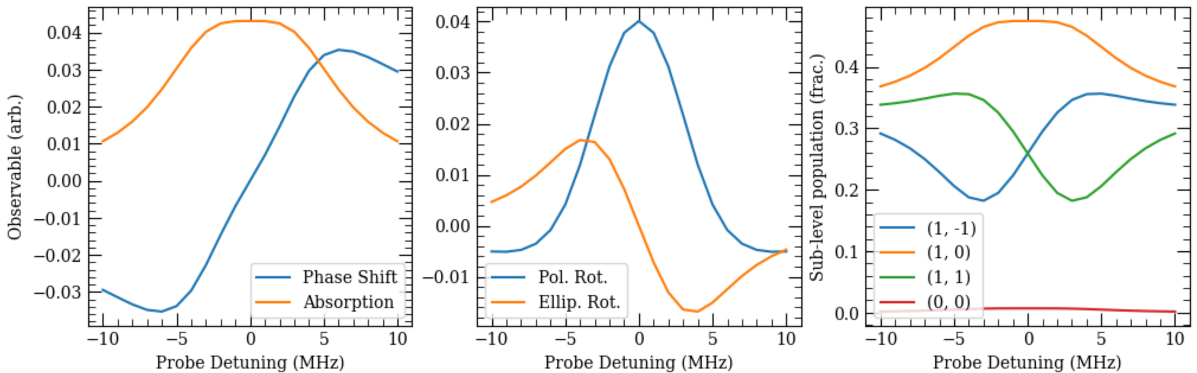
```
aligned_obs = solF1F0sig.coupling_coefficient_matrix(((1, 'all'), e))
perp_obs = np.abs(aligned_obs)
```

```
susc_aligned = solF1F0sig.coupling_coefficient_observable(((1, 'all'), e))
susc_perp = solF1F0sig.get_observable(perp_obs)
```

```
pops = rq.get_rho_populations(solF1F0sig)
```

```
fig, (ax, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3.5))

ax.plot(dets, susc_aligned.real, label='Phase Shift')
ax.plot(dets, susc_aligned.imag, label='Absorption')
ax.legend()
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('Observable (arb.)')

ax2.plot(dets, susc_perp.real,label='Pol. Rot.')
ax2.plot(dets, susc_perp.imag, label='Ellip. Rot.')
ax2.legend()
ax2.set_xlabel('Probe Detuning (MHz)')

ax3.plot(dets, pops, label=s.states)
ax3.legend()
ax3.set_xlabel('Probe Detuning (MHz)')
ax3.set_ylabel('Sub-level population (frac.)')
```

```
Text(0, 0.5, 'Sub-level population (frac.)')
```



13.4 A F=1 to F'=0 spectroscopy experiment with $\sigma^{\pm}$ and a static detuning and a varying axial magnetic field

This is the prototypical NMOR experiment. A linearly-polarized probing field interacts with the atomic system. The quantization axis is aligned with the axial magnetic field, meaning the probing light is decomposed into equal parts of left and right circular polarization. Sweeping the axial magnetic field strength produces nested dispersive features in the polarization rotation of the probe field. The larger feature is linear birefringence due to un-even absorption of the two polarization components. The narrow feature is NMOR, resulting from a non-linear interaction

```
g = (1, [-1,0,1])
e = (0, 0)
```

```
s = rq.Sensor([g,e])
```

```
det = 3
rabi_freq = 1
cg = {((1,-1), e):1,
      ((1,0), e):0,
      ((1,1), e):-1}

s.add_coupling((g,e), rabi_frequency=2*np.pi*rabi_freq, detuning=2*np.pi*det,␣
↪coupling_coefficients=cg, label='probe')
```

```
cgg = {(e, (1,-1)): 1/3,
       (e, (1,0)): 1/3,
       (e, (1,1)): 1/3}
s.add_decoherence((e, g), 2*np.pi*6.0666, coupling_coefficients=cgg, label='F0_
↪gamma')

s.add_transit_broadening(2*np.pi*0.1, repop={state:1/3 for state in s.states_with_
↪spec(g)})
```

```
larmor_freqs = np.linspace(-15, 15, 301)
prefactors = {(1,-1): 2*np.pi,
              (1,1): -2*np.pi}

s.add_energy_shift(g, larmor_freqs, prefactors=prefactors)
```

```
solF1F0sigNMOR = rq.solve_steady_state(s)
```

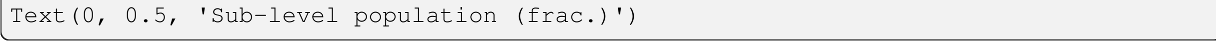
```
aligned_obs = solF1F0sigNMOR.coupling_coefficient_matrix(((1, 'all'), e))
print(aligned_obs)
perp_obs = np.abs(aligned_obs)
print(perp_obs)
```

```
[[ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 0.   0.   0.   0.]
 [ 1.   0.  -1.   0.]]
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [1. 0. 1. 0.]]
```

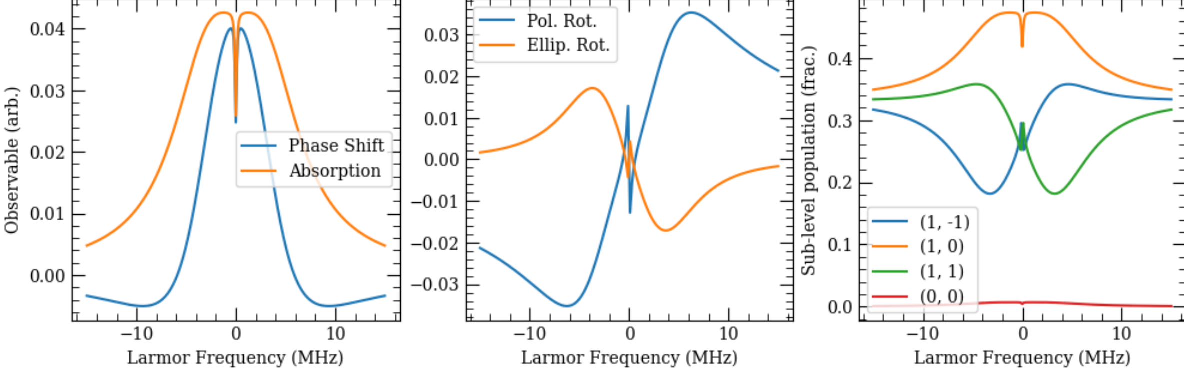
```
susc_aligned = solF1F0sigNMOR.coupling_coefficient_observable(((1, 'all'), e))
susc_perp = solF1F0sigNMOR.get_observable(perp_obs)
```

```
pops = rq.get_rho_populations(solF1F0sigNMOR)
```

```
fig, (ax, ax2, ax3) = plt.subplots(1, 3, figsize=(12, 3.5))

ax.plot(larmor_freqs, susc_aligned.real, label='Phase Shift')
ax.plot(larmor_freqs, susc_aligned.imag, label='Absorption')
ax.legend()
ax.set_xlabel('Larmor Frequency (MHz)')
ax.set_ylabel('Observable (arb.)')

ax2.plot(larmor_freqs, susc_perp.real,label='Pol. Rot.')
ax2.plot(larmor_freqs, susc_perp.imag, label='Ellip. Rot.')
ax2.legend()
ax2.set_xlabel('Larmor Frequency (MHz)')

ax3.plot(larmor_freqs, pops, label=s.states)
ax3.legend()
ax3.set_xlabel('Larmor Frequency (MHz)')
ax3.set_ylabel('Sub-level population (frac.)')
```



```
Text(0, 0.5, 'Sub-level population (frac.)')
```



13.5 A Rb87 D2 spectroscopy experiment with $\sigma^{\pm}$ polarized light

This is equivalent to doing D2 spectroscopy of atoms in a cold beam of Rubidium87 atoms. The transit of the beam prevents all the atoms from being trivially optically pumped to the stretch state.

To do this correctly, we need to properly scale Rabi frequencies and dephasings by the actual Clebsch-Gordon coefficient for each sub-level transition.

We leverage ARC's implementation to provide this functionality here.

`CG(F, mF, 1, q, F', mF')`

```
Fps = [0,1,2,3]
Fgs = [1,2]
e_states = [[('e',F,mF) for mF in range(-F,F+1)] for F in Fps]
print(e_states)
g_states = [[('g',F,mF) for mF in range(-F,F+1)] for F in Fgs]
print(g_states)
e_spec = ('e','all','all')
g_spec = ('g','all','all')
```

```
[[('e', 0, 0)], [('e', 1, -1), ('e', 1, 0), ('e', 1, 1)], [('e', 2, -2), ('e', 2, -
↪1), ('e', 2, 0), ('e', 2, 1), ('e', 2, 2)], [('e', 3, -3), ('e', 3, -2), ('e', 3,
↪ -1), ('e', 3, 0), ('e', 3, 1), ('e', 3, 2), ('e', 3, 3)]]
[[('g', 1, -1), ('g', 1, 0), ('g', 1, 1)], [('g', 2, -2), ('g', 2, -1), ('g', 2,␣
↪0), ('g', 2, 1), ('g', 2, 2)]]
```

```
# this flattens the list of lists
e_list = list(itertools.chain.from_iterable(e_states))
g_list = list(itertools.chain.from_iterable(g_states))
```

```
s = rq.Sensor(g_list+e_list)
print(s.states)
print(f'There are {len(s.states):d} total states')
```

```
[('g', 1, -1), ('g', 1, 0), ('g', 1, 1), ('g', 2, -2), ('g', 2, -1), ('g', 2, 0), (
↪'g', 2, 1), ('g', 2, 2), ('e', 0, 0), ('e', 1, -1), ('e', 1, 0), ('e', 1, 1), ('e
↪', 2, -2), ('e', 2, -1), ('e', 2, 0), ('e', 2, 1), ('e', 2, 2), ('e', 3, -3), ('e
↪', 3, -2), ('e', 3, -1), ('e', 3, 0), ('e', 3, 1), ('e', 3, 2), ('e', 3, 3)]
There are 24 total states
```

We iterate through all optical couplings, ignoring those that are not dipole allowed (ie CG=0).

```
qs = [-1, 1]
cg = {(('g',Fg,mFg), ('e',Fe,mFe)): CG(Fg, mFg, 1, q, Fe, mFe)**2
      for Fg in Fgs
      for Fe in Fps
      for mFg in range(-Fg,Fg+1)
      for mFe in range(-Fe, Fe+1)
      for q in qs
      if CG(Fg, mFg, 1, q, Fe, mFe) != 0.0}
print(f'There are {len(cg):d} total couplings')
```

```
There are 36 total couplings
```

```
dets = np.linspace(-310, 205, 231)
rabi_freq = 1
```

(continues on next page)

(continued from previous page)

```
s.add_coupling((g_spec, e_spec), rabi_frequency=2*np.pi*rabi_freq, detuning=2*np.
↪pi*dets, coupling_coefficients=cg, label='probe')
```

We iterate through all possible dephasing paths (including those that don't have optical couplings on them), again ignoring those that are not dipole-allowed.

```
qs = [-1, 0, 1]
cgg = {(('e',Fe,mFe), ('g',Fg,mFg)): CG(Fg, mFg, 1, q, Fe, mFe)**2
       for Fg in Fgs
       for Fe in Fps
       for mFg in range(-Fg,Fg+1)
       for mFe in range(-Fe, Fe+1)
       for q in qs
       if CG(Fg, mFg, 1, q, Fe, mFe) != 0.0}
print(f'There are {len(cgg):d} total lifetime dephasings')
```

```
There are 54 total lifetime dephasings
```

```
s.add_decoherence((e_spec,g_spec), 2*np.pi*6.0666, coupling_coefficients=cgg,␣
↪label='HFS_gamma')
```

We add some minor transit broadening. In a doppler-free solution, this would be equivalent to a cold beam of atoms with a transit time through the beams equivalent to 100kHz.

```
s.add_transit_broadening(2*np.pi*0.1, repop={state: 1/len(g_list) for state in g_
↪list})
```

We now add the hyperfine shifts of the various manifolds. The excited state is defined relative to the center of mass energy (i.e. fine structure only). The ground states are defined relative to our primary state of interest.

```
hfs_shifts = {**{state: 2*np.pi*193.7407 for state in s.states_with_spec(('e',3,
↪'all'))},
             **{state: -2*np.pi*72.9112 for state in s.states_with_spec(('e',2,'all
↪'))},
             **{state: -2*np.pi*229.8518 for state in s.states_with_spec(('e',1,
↪'all'))},
             **{state: -2*np.pi*302.0738 for state in s.states_with_spec(('e',0,
↪'all'))},
             **{state: -2*np.pi*6834.682610904290 for state in s.states_with_spec((
↪'g',1,'all'))}
             }
```

```
s.add_energy_shifts(hfs_shifts)
```

```
%%time
solRb87D2 = rq.solve_steady_state(s)
```

```
CPU times: total: 33.2 s
Wall time: 8.66 s
```

```
susc = solRb87D2.coupling_coefficient_observable((e_spec, g_spec))
```

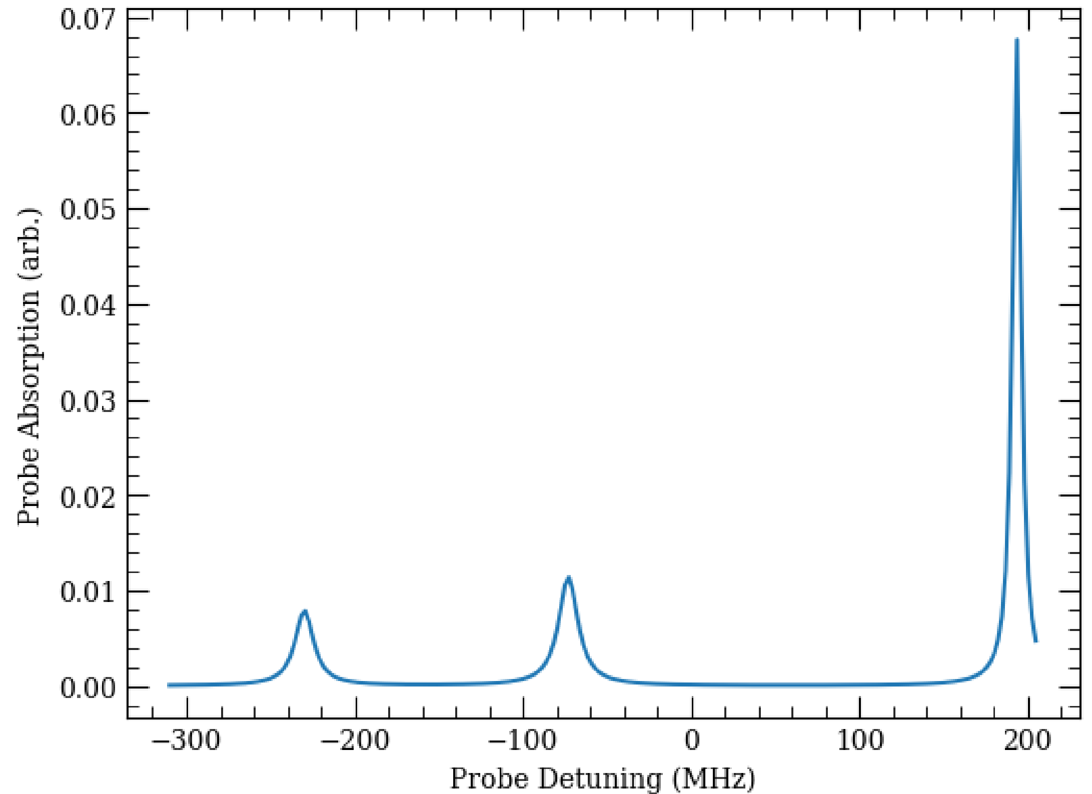
```
fig, ax = plt.subplots(1)
```

(continues on next page)

(continued from previous page)

```
ax.plot(dets, susc.imag)
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('Probe Absorption (arb.)')
```

```
Text(0, 0.5, 'Probe Absorption (arb.)')
```




```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     1.3.0.dev474+g005aa83.d20241205
Installation Path:    ~\src\rydiqule\src\rydiqule

      Dependencies
    ================

NumPy Version:        1.26.4
SciPy Version:        1.10.1
Matplotlib Version:   3.7.1
ARC Version:          3.6.0
Python Version:       3.11.5
Python Install Path:  ~\miniconda3\envs\rq
Platform Info:        Windows (AMD64)
CPU Count and Freq:   16 @ 3.91 GHz
Total System Memory:  256 GB
```

rydiqule

CHAPTER FOURTEEN

RF HETERODYNE WITH DOPPLER EXAMPLE

This notebook demonstrates two-tone detection using a Rydberg sensor in the time domain with Doppler averaging. An RF local oscillator (LO) and signal (sig) are imposed on the Rydberg sensor. This is useful for RF phase detection, and can be used to linearize the detection, as shown below. The main results of this example showing how different levels of Doppler averaging affect the beat signal size of the sensor.

This notebook can be downloaded here.

```
import datetime
##***LAST UPDATE***##
now = datetime.datetime.now()
print(now)
```

```
2026-01-20 09:34:52.435245
```

14.1 Imports

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

14.2 Define the Sensors

```
atom = "Rb85"
(g, e) = rq.D2_states(atom)
r1 = rq.A_QState(150, 2, 2.5)
r2 = rq.A_QState(149, 3, 3.5)
```

```
rf_rabi = 100 #Mrad/s
red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*5}  #fields are stored as␣
↪dictioniaries
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*7, 'detuning': 0}
LO_ss = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'detuning':0}


RbSensor_ss = rq.Cell(atom, [g, e, r1, r2],
                      gamma_transit=2*np.pi*1, cell_length = 0.01)
RbSensor_time = rq.Cell(atom, [g, e, r1, r2],
                      gamma_transit=2*np.pi*1, cell_length = 0.01)
```

```
state1 = RbSensor_time.states[2]
state2 = RbSensor_time.states[3]
```

(continues on next page)

(continued from previous page)

```
print("1: ", state1)
print("2: ", state1)
dipoleMoment = RbSensor_time.atom.get_dipole_matrix_element(state1,state2, 0)

field = rf_rabi/rq.scale_dipole(dipoleMoment)

print("applied field, V/m:", field) #V/m
print("Rabi frequency, Mrad/s: ", field*rq.scale_dipole(dipoleMoment))
```

```
1:  (150, 2, 2.5)
2:  (150, 2, 2.5)
applied field, V/m: 0.09799930011573187
Rabi frequency, Mrad/s:  100.0
```

```
def sig_and_LO( delta, beta):
    def fun(t):
        return (1+beta*np.sin(delta*t))
    return fun
```

```
rf_freq = RbSensor_time.atom.arc_atom.getTransitionFrequency(*r1[:3],*r2[:3])*1E-6
rf_freq #MHz
```

```
658.5872654159832
```

14.3 Observe a heterodyne beat between the Signal and LO.

14.3.1 Define the RF LO and signal

```
sampleNum = 200
endTime = 10 # microseconds
rf = sig_and_LO( 5, .1)
```

14.3.2 Solve without Doppler averaging

Observe the beat between signal and LO fields.

```
red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*5, 'detuning':0}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*7, 'detuning': 0}
rf = {'states':(r1,r2), "rabi_frequency": rf_rabi, 'detuning': 0, 'time_dependence
↪': sig_and_LO( 2*np.pi, .05)}

RbSensor_time.add_couplings(blue_laser, red_laser, rf)
```

```
#Solve Without any doppler broadening

time_sol = rq.solve_time(RbSensor_time, endTime, sampleNum, atol=1e-6, rtol=1e-6)
```

```
transmission = time_sol.get_transmission_coef()
```

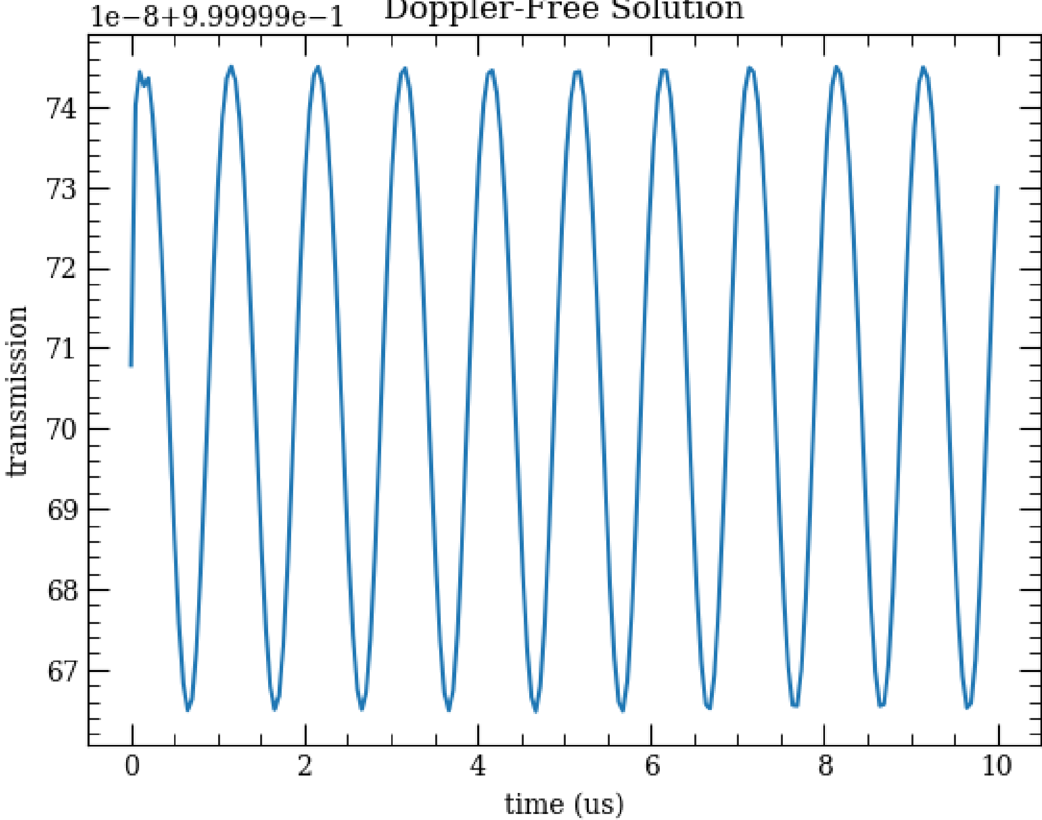
```
fig, ax = plt.subplots()
ax.plot(time_sol.t, transmission)
ax.set_xlabel("time (us)")
```

(continues on next page)

(continued from previous page)

```
ax.set_ylabel('transmission')
ax.set_title("Doppler-Free Solution")
```

```
Text(0.5, 1.0, 'Doppler-Free Solution')
```



14.3.3 Solve with Doppler averaging

Doppler averaged results require larger Rabi frequencies to observe similar sized signals.

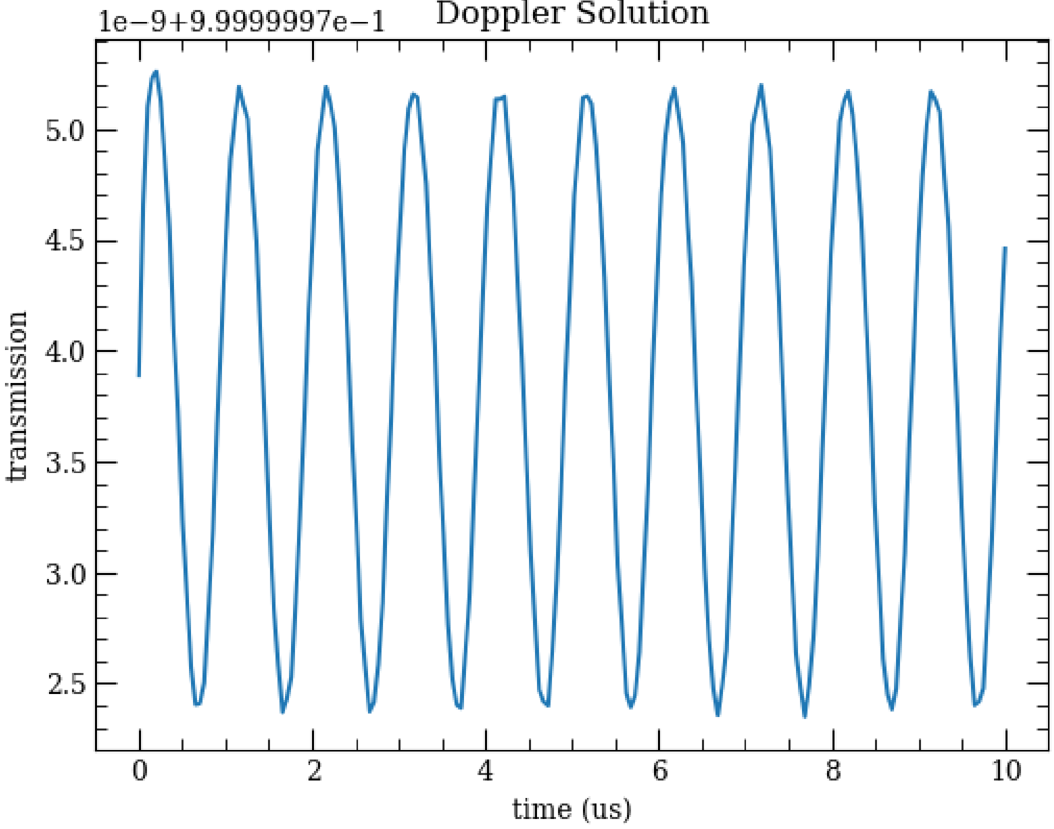
```
red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*5, 'detuning':0, 'kunit': np.
↪array([1,0,0])}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*7, 'detuning': 0,'kunit':␣
↪np.array([-1,0,0])}
rf = {'states':(r1,r2), "rabi_frequency":rf_rabi, 'detuning': 0, 'time_dependence
↪': sig_and_LO( 2*np.pi, .05)}

RbSensor_time.add_couplings(blue_laser, red_laser, rf)
```

```
#Solve with a doppler peak calculated from physical system properties
sampleNum = 200
endTime = 10
time_sol_doppler = rq.solve_time(RbSensor_time, endTime, sampleNum, doppler=True,␣
↪rtol = 1e-6, atol = 1e-6)
```

```
transmission_doppler = time_sol_doppler.get_transmission_coef()
```

```
fig, ax = plt.subplots()
ax.plot(time_sol_doppler.t, transmission_doppler)
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission')
ax.set_title("Doppler Solution")
```

```
Text(0.5, 1.0, 'Doppler Solution')
```



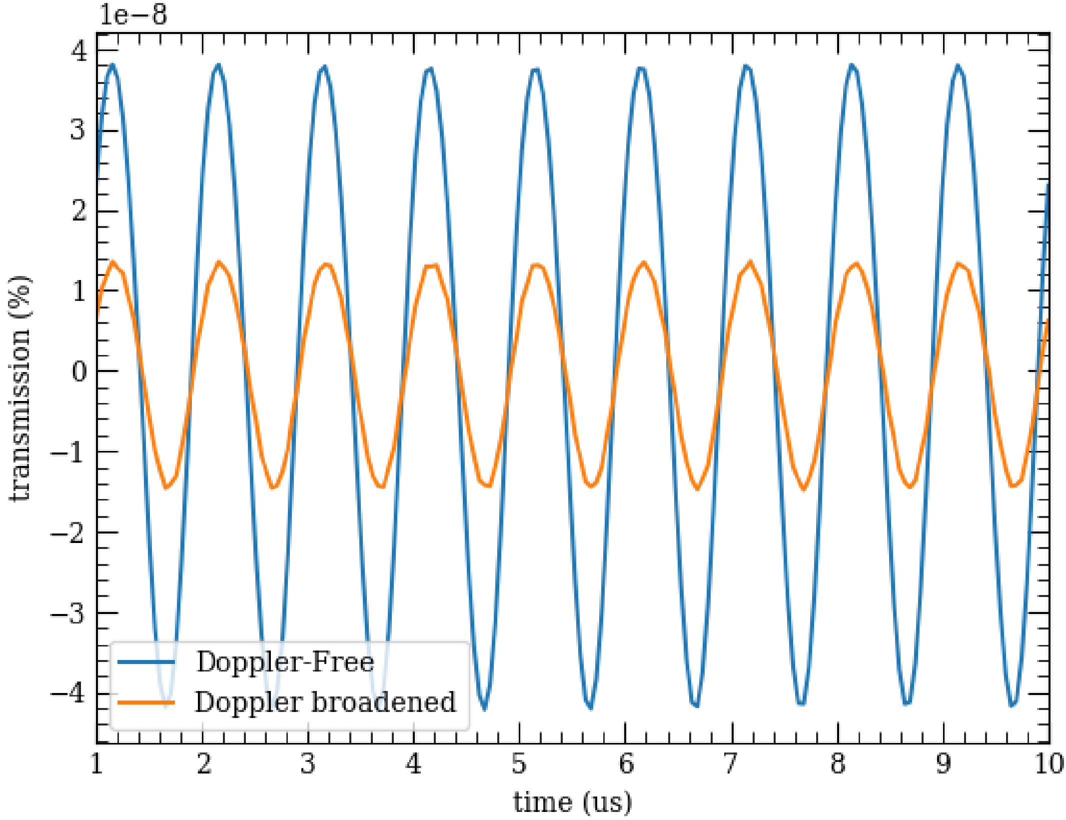
14.3.4 Compare the size of the beat signals

Here we ignore the starting transient, and normalize the beat signal. As the Doppler broadening is increased, the size of the beat is reduced (for the same optical depth).

```
def normalize_trace(trace,expand=1):
    ave = trace[100:].mean()
    return (trace - ave)/ave*expand
```

```
fig, ax = plt.subplots()

ax.plot(time_sol.t, normalize_trace(transmission), label='Doppler-Free')
ax.plot(time_sol_doppler.t, normalize_trace(transmission_doppler,10), label=
↪'Doppler broadened')
ax.set_xlim((1,10))
ax.set_xlabel("time (us)")
ax.set_ylabel('transmission (%)')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x14ec1d99b40>
```




```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.2.0.dev48+g6f06f9960.d20260120
Installation Path:    ~\src\rydiqule_public\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.2.6
SciPy Version:        1.15.3
Matplotlib Version:   3.10.8
ARC Version:          3.9.0
Python Version:       3.10.19
Python Install Path:  ~\src\rydiqule_public\.venv\Scripts
Platform Info:        Windows (AMD64)
CPU Count and Freq:   16 @ 3.91 GHz
Total System Memory:  256 GB
```

CHAPTER

FIFTEEN

RF HETERODYNE EXAMPLE

For a more thorough introduction to the core functionality of rydiqule, it may be helpful to look at the Introduction_to_Rydiqule.ipynb notebook before this one.

This notebook demonstrates two-tone detection using a Rydberg sensor in the time domain. An RF local oscillator (LO) and signal (sig) are imposed on the Rydberg sensor. This is useful for RF phase detection, and can be used to linearize the detection, as shown below. The main results of this example are:

1. we show that the time solver and steady-state solver approximately agree.
2. we plot an example of a time response due to RF heterodyne.
3. we find the optimum detuning of the Rydberg laser for RF heterodyne, for a given value of LO power.
4. we plot the linear dynamic of the RF heterodyning scheme, and show that it is limited by the LO power on the high end. The result is limited by the solver tolerance on the low end.

This notebook can be downloaded here.

15.1 Imports

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

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

15.2 Comparing the steady-state and time solver results

This example uses a Cell object, which inherits Sensor. The Cell's purpose is to attach the bare pyhiscs calculations of a Sensor to a real physical atom. This allows for specification of quantum numbers for states, meaning rydiqule can calculate things like transition frequencies (using ARC) without needing to specify them in the object creation. The details will be discussed further down.

NOTE: The time solver runs more slowly for large transition frequencies, since it makes no rotating wave approximation by default. Therefore, it is advisable to debug calculations using a transition with a low frequency (ie, very large n). Once calculations are debugged and running well, they can be re-run with the appropriate n-level. Further, ARC calculates dipole moments for the chosen transition. For large n, this calculation is slow, due to the large amount of structure in the atomic wavefunction. However, ARC caches the results, so it only runs slowly the first time.

15.2.1 The steady-state Cell

```
atom = "Rb85"

(g, e) = rq.D2_states(atom)
```

(continues on next page)

(continued from previous page)

```
r1 = rq.A_QState(150, 2, 2.5)
r2 = rq.A_QState(149, 3, 3.5)


RbSensor_ss = rq.Cell(atom, [g, e, r1, r2],
                      gamma_transit=2*np.pi*1, cell_length = 1e-5)
```

Define Transitions

Transitions are defined as dictionaries in a `Cell` in the same way as they are in a bas `Sensor`. For the steady-state case, nothing changes. For this example, we will observe the response of the system over a series of 200 blue laser detunings.

```
rf_rabi = 25 #Mrad/s
n_det_ss = 200
detunings_ss = np.linspace(-150, 150, n_det_ss)

red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning':detunings_
↪ss}
local_oscillator_ss = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'detuning':0}

RbSensor_ss.add_couplings(red_laser, blue_laser, local_oscillator_ss)
```

Solve the steady state system

We solve a `Cell` in exaclty the same way we solve a `Sensor` object.

```
ss_solution = rq.solve_steady_state(RbSensor_ss)
print(ss_solution.rho.shape)
```

```
(200, 15)
```

15.2.2 The time solver `Cell`

We want to use the same decoherence values

```
RbSensor_time = rq.Cell(atom, [g, e, r1, r2],
                        gamma_transit=2*np.pi*1, cell_length = 1e-5)
```

Defining the rf field

The `time_dependence` argument is expected to be a python function of a single variable (time in μs) that returns the field at that time. To match our steady state solution, which had a detuning of 0, we will explicitly define a single-tone field as a function of time that is resonant with our rf transition.

```
rf_freq = RbSensor_ss.atom.get_transition_frequency(r1, r2)*1E-6
def rf_carrier(t):
    return np.cos(2*np.pi*rf_freq*t)#extra factor of 2 to account for no RWA.
```

```
n_det = 20
detunings = np.linspace(-75, 75, n_det)

red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning':detunings}
```

(continues on next page)

(continued from previous page)

```
rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'time_dependence': rf_
↪carrier }

RbSensor_time.add_couplings(red_laser, blue_laser, rf_transition)
```

Solve in the time domain

```
%%time
end_time = 10 #microseconds
sample_num = 10

time_solution = rq.solve_time(RbSensor_time, end_time, sample_num, atol=1e-6,␣
↪rtol=1e-6)
```

```
CPU times: total: 11.2 s
Wall time: 11.3 s
```

```
RbSensor_time.couplings.edges[r1,r2]
```

```
{'rabi_frequency': 25,
 'transition_frequency': 4138.0258295573,
 'kvec': (0, 0, 0),
 'time_dependence': <function __main__.rf_carrier(t)>,
 'coherent_cc': 1,
 'dipole_moment': np.float64(12692.33892452404),
 'q': 0,
 'label': '((150, 2, 2.5),(149, 3, 3.5))'}
```

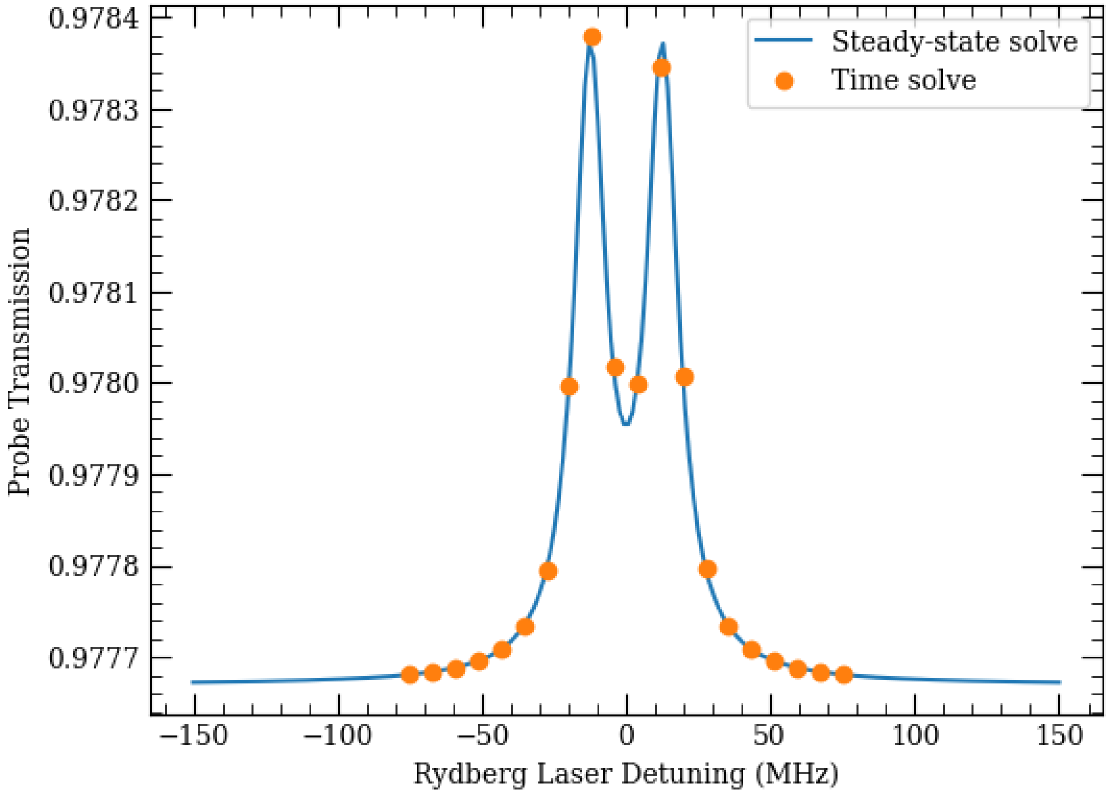
15.2.3 Comparing results

Now, with results for both steady state and time simulations, we can compare them and see that they match. Note that because the time solution has an extra dimension, we only get the last (`[:,-1]`) element, since this is effectively the steady-state solution (assuming transient behavior has been damped out by 10 μs). Here we also use a function to get the transmission coefficient from the solution quickly by extracting the proper density matrix elements.

```
time_solution.get_transmission_coef().shape
```

```
(20, 10)
```

```
#Modify to include convenience functions and get physical parameters.

fig, ax = plt.subplots()
ax.plot(detunings_ss, ss_solution.get_transmission_coef(), label="Steady-state␣
↪solve")
ax.plot(detunings, time_solution.get_transmission_coef()[:,-1], 'o', label="Time␣
↪solve")
ax.set_xlabel("Rydberg Laser Detuning (MHz)")
ax.set_ylabel("Probe Transmission")
ax.legend();
```



15.3 Observe a heterodyne beat between the Signal and LO.

The goal of this section is to see the beating behavior in time between the signal and rf local oscillator. This will help see exactly how to observe behavior in the time domain with rydiqule. It will look a lot like other time solves, but we will look at the solution in a different way.

15.3.1 Define the RF LO and signal

Just like before, we need a function of time to input into the time solver. This time, instead of just an rf carrier signal, we will define a function that adds a local oscillator of frequency ω_0 with an "incoming" rf signal of frequency $\omega_0 + \delta$. We will also define the time function as the return of another function, which will allow us to make changes to the function quickly if we want to experiment a little.

```
def sig_and_LO(omega_0, delta, beta):
    def fun(t):
        return np.sin(omega_0*t)+beta*np.sin((omega_0+delta)*t)
    return fun
```

```
omega_0 = 2*np.pi*rf_freq
delta = 5
beta = 0.05
```

15.3.2 The Sensor

We will use the same `RbSensor_time` sensor as before, but change the blue laser to be just a single value. This highlights an important aspect of `Sensor`. It does not support multiple fields coupling the same pair of levels, but will override an old one with a new one when `add_coupling()` is called.

```
red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning': 0}
rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'time_dependence':
↪sig_and_LO(omega_0, delta, beta )}
RbSensor_time.add_couplings(red_laser, blue_laser, rf_transition)
```

15.3.3 Inital conditions

If the `init_cond` argument is not supplied to `rq.solve_time`, it will calculate the initial condition based on the steady-state solution of the supplied sensor with the $t = 0$ time-dependant field values. Since we define our incoming field and LO in a single function, we will likely end up with a sizeable transient if we use this approach if the $t = 0$ value happens to be 0. This is now a great opportunity to demonstrate how to supply an initial condition manually. We calculate our initial condition using the solution to the **steady-state** sensor we defined above. Hopefully, this allows us to see the beat oscillation around the steady-state solution without a large transient from introducing our LO and rf field at the same time.

Solving is done the same way as before, this time solving for 250 points over 10 microseconds

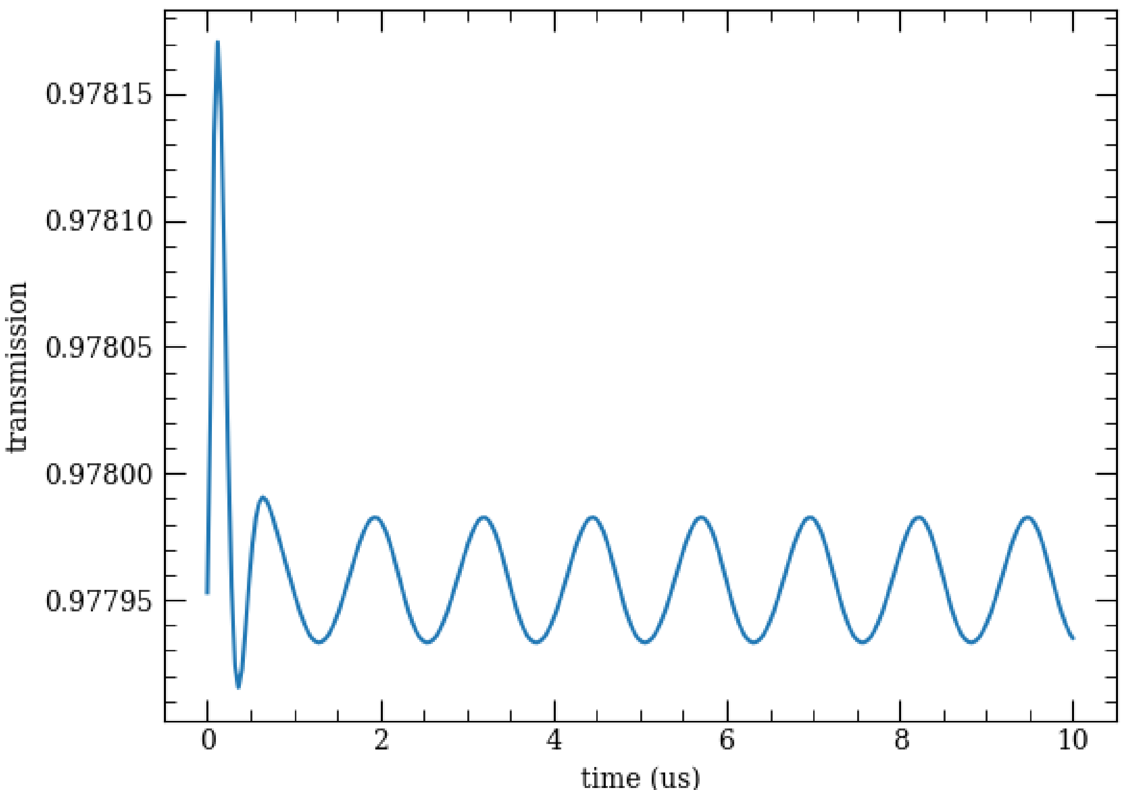
```
RbSensor_ss.add_couplings(blue_laser)
sol_init = rq.solve_steady_state(RbSensor_ss)

sample_num=250
end_time = 10

time_sol_beat = rq.solve_time(RbSensor_time, end_time, sample_num,  init_cond=sol_
↪init.rho)

print(time_sol_beat.rho.shape)
```

```
(250, 15)
```

15.3.4 Plotting the beat

We now use the `get_transmission_coef()` function again to extract the transmission from the `(250, 15)`-shaped solution to get a 250 element array we can plot against time. We can see that the system quickly settles into the expected beat frequency.

```
fig, ax = plt.subplots()
transmission = time_sol_beat.rho[:,3]
ax.plot(time_sol_beat.t, time_sol_beat.get_transmission_coef())
ax.set_xlabel("time (us)");
ax.set_ylabel('transmission');
```

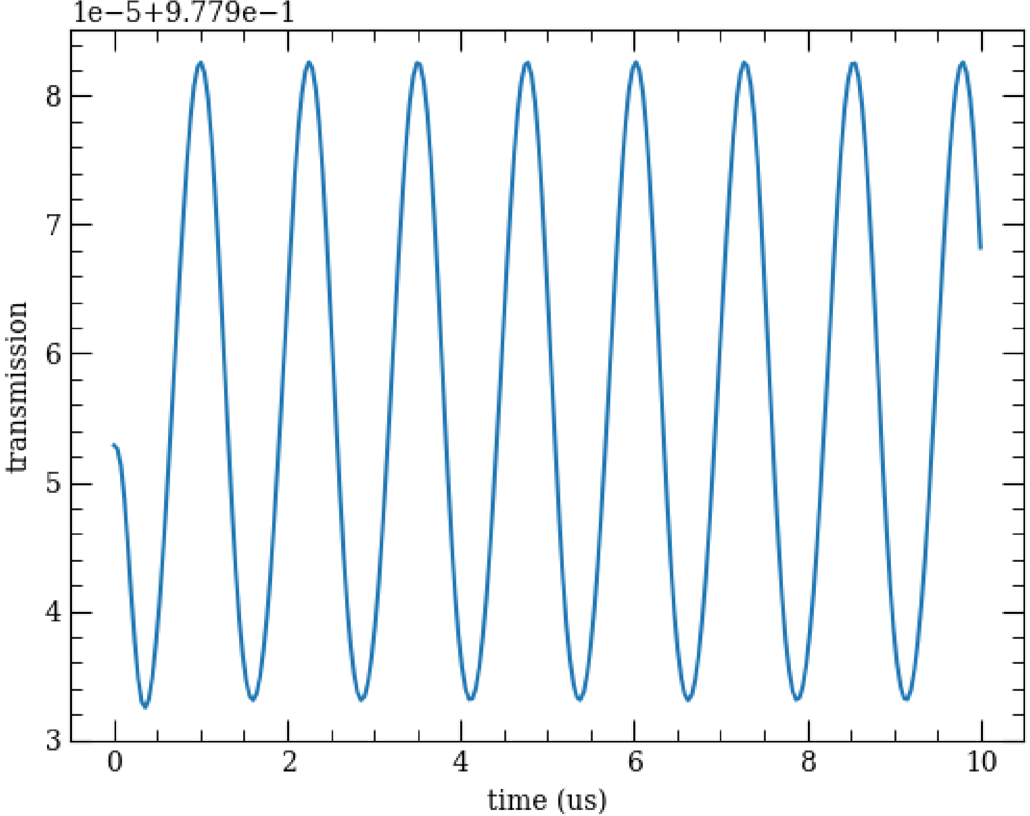


We do have a bit of transient behavior at the start, but we can clearly see the expected beat frequency of 5Mrad/s $\approx$ 800kHz. The transient behavior could be removed by modifying the time-dependence function to return the LO amplitude at $t = 0$, since rydiqule uses the $t = 0$ values of time-dependent functions when solving the initial-conditions.

```
def sig_and_LO_shifted(omega_0, delta, beta):
    def fun(t):
        return np.cos(omega_0*t)+beta*np.sin((omega_0+delta)*t)  # note LO phase␣
↪changed 90deg by changing from sin to cos
    return fun
```

```
rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'time_dependence':␣
↪sig_and_LO_shifted(omega_0, delta, beta )}
RbSensor_time.add_couplings(rf_transition)
time_sol_beat2 = rq.solve_time(RbSensor_time, end_time, sample_num, init_cond=sol_
↪init.rho)
```

```
fig, ax = plt.subplots()
transmission = time_sol_beat2.rho[:,3]
ax.plot(time_sol_beat2.t, time_sol_beat2.get_transmission_coef())
ax.set_xlabel("time (us)");
ax.set_ylabel('transmission');
```



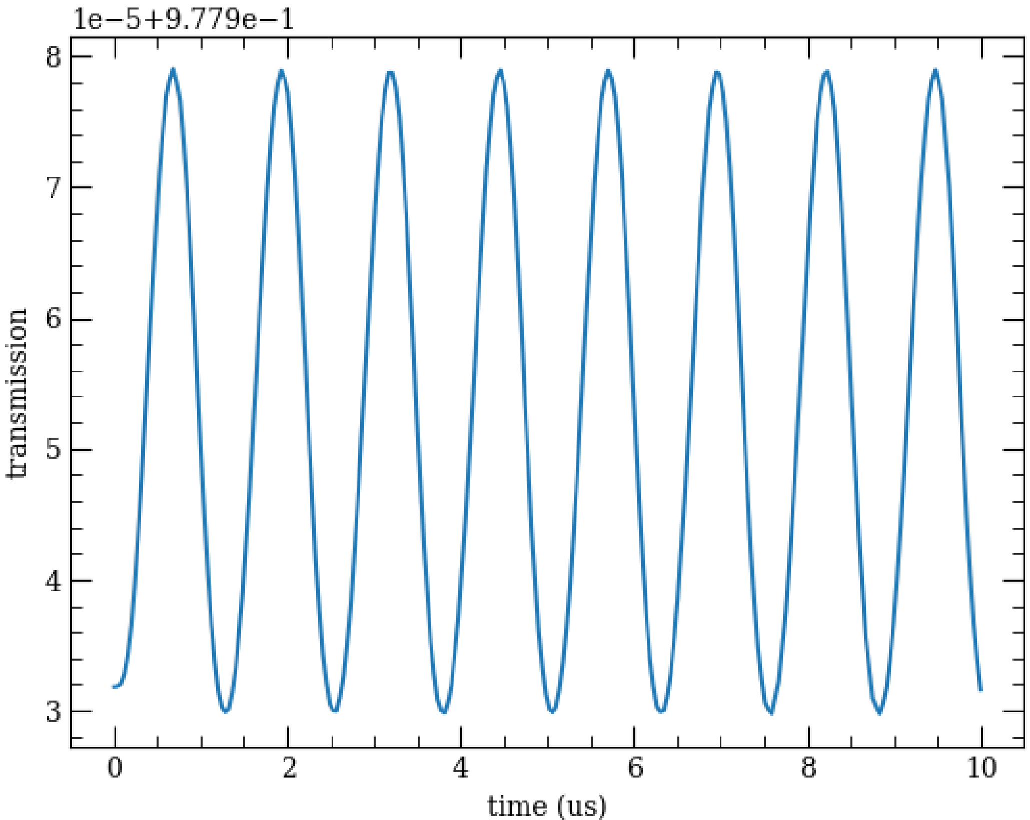
15.4 RF Heterodyne in the Rotating Wave Approximation

We move to a rotating frame by specifying an rf detuning, and writing the coupling in the complex rotating frame. This transformation will greatly speed up the time integration.

```
def sig_LO_RWA(det, beta):
    def fun(t):
        return 1+beta*np.exp(1j*det*t)
    return fun

red_laser = {'states':(g,e), 'rabi_frequency':2*np.pi*0.6, 'detuning':0}
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning': 0}
rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'detuning':0, 'time_
↪dependence': sig_LO_RWA(delta, beta )}
RbSensor_time.add_couplings(red_laser, blue_laser, rf_transition)

sol_init = rq.solve_steady_state(RbSensor_ss)
```

```
sample_num=250
end_time = 10
time_sol_beat = rq.solve_time(RbSensor_time, end_time, sample_num)
```

```
fig, ax = plt.subplots()
ax.plot(time_sol_beat.t, time_sol_beat.get_transmission_coef())
ax.set_xlabel("time (us)");
ax.set_ylabel('transmission');
```



15.5 Find the optimum laser detuning with LO

We again use the same sensor to look at the sensitivity of the sensor as a function of blue laser detuning.

15.5.1 Set up and solve Sensor

```
num_dets = 75
detuning_list = np.linspace(-40,40,num_dets)
pk_to_pk_result = np.zeros(num_dets)

sample_num = 300
end_time = 3
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning': detuning_
↪list}
rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'time_dependence':␣
↪sig_and_LO(rf_freq*2*np.pi, 5, 0.01 )}
RbSensor_time.add_couplings(blue_laser, rf_transition) #this replaces the old␣
↪coupling

time_sol = rq.solve_time(RbSensor_time, end_time, sample_num)
```

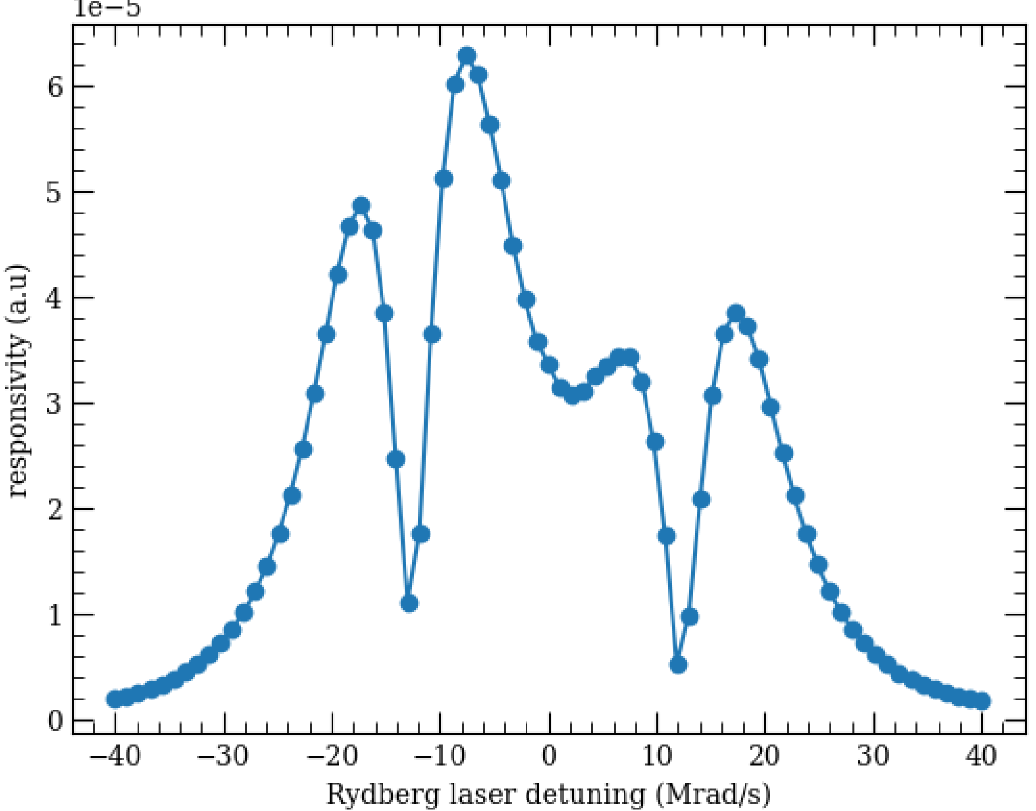
```
susceptibility = time_sol.rho[:,100:,3]
print(susceptibility.shape)
ptp_result = np.ptp(susceptibility, axis=-1)
```

```
(75, 200)
```

15.5.2 Plotting responsivity versus LO detuning

```
fig, ax = plt.subplots()
ax.plot(detuning_list, ptp_result,'o-')
ax.set_ylabel("responsivity (a.u)")
ax.set_xlabel("Rydberg laser detuning (Mrad/s)")
```

```
Text(0.5, 0, 'Rydberg laser detuning (Mrad/s)')
```



This plot shows that, for maximum responsivity, the Rydberg (or probe) laser must be tuned to the side of an Autler-Townes peak, for maximum sensitivity

15.6 Test the Linear Dynamic Range

Example testing the linear dynamic range in Heterodyne

15.6.1 Setting the laser parameters

We will set the blue laser detuning to -8 MRad/s, which was roughly the optimal value from the plot above

```
blue_laser = {'states':(e,r1), 'rabi_frequency':2*np.pi*1.0, 'detuning': -8}
RbSensor_time.add_couplings(blue_laser)
```

15.6.2 Solve parameters

With the detuning set, we can set up everything we need for our scan, namely the solver parameters and list of amplitudes.

```
num_Amps = 50
amp_list = np.logspace(-6,0.2,num_Amps)
sample_num = 300
end_time = 3
```

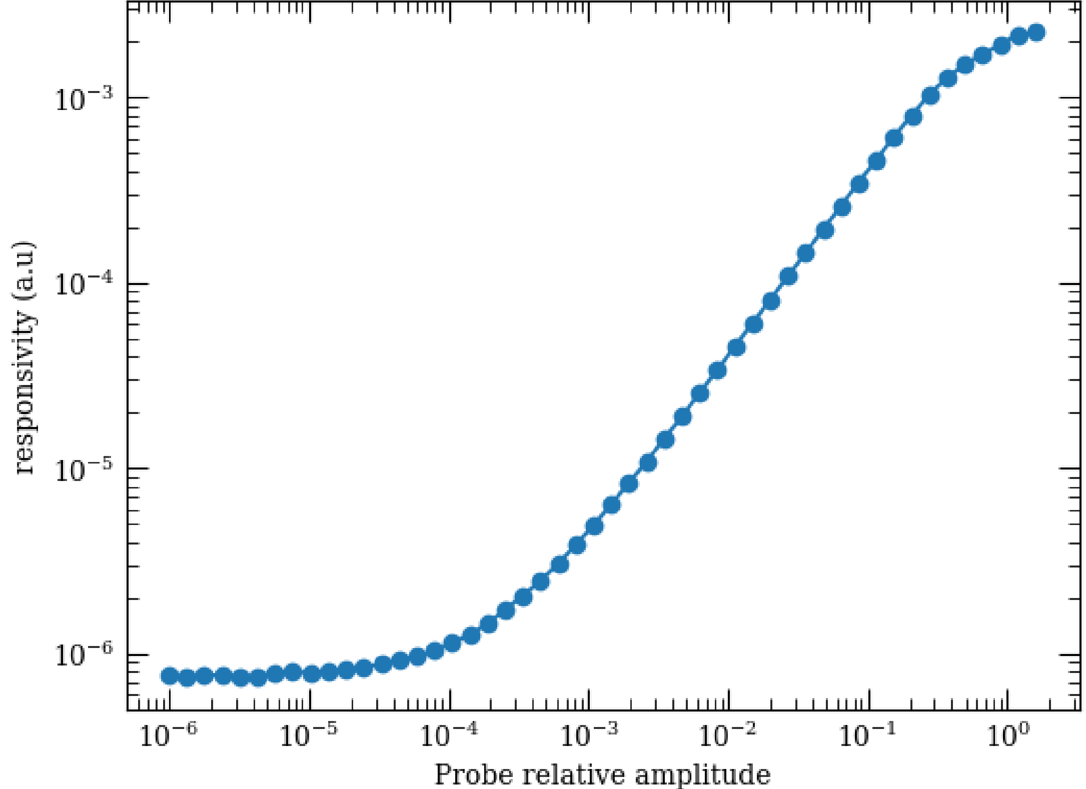
15.6.3 Running the loop

Now all that is left is get the value we want at each amplitude using a good old python `for` loop. This could be done with something like a `map()` function depending on your comfort level with python, but we will be explicit here.

```
pk_to_pk_result = np.zeros(num_Amps)

for idx, amp in enumerate(amp_list):
    #define and solve for rf input

    rf = sig_and_LO(2*np.pi*rf_freq, 5, amp)
    rf_transition = {'states':(r1,r2), 'rabi_frequency':rf_rabi, 'detuning':1,
↪'time_dependence': sig_LO_RWA(1, amp)}
    RbSensor_time.add_couplings(rf_transition)
    time_sol = rq.solve_time(RbSensor_time, end_time, sample_num,  init_cond=sol_
↪init.rho, atol=1e-7, rtol=1e-7)

    #calculate responsivity
    pk_to_pk_signal = np.ptp(time_sol.rho[100:,3])
    pk_to_pk_result[idx] = pk_to_pk_signal
```

15.6.4 Plotting the dynamic range

```
fig, ax = plt.subplots()
ax.plot(amp_list, pk_to_pk_result,'o-')
ax.set_ylabel("responsivity (a.u)")
ax.set_xlabel("Probe relative amplitude")
ax.set_xscale('log')
ax.set_yscale('log')
```




```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.2.0.dev31+g00cba9d8.d20250823
Installation Path:    ~\src\rydiqule_public\src\rydiqule

      Dependencies
    ================

NumPy Version:        2.2.5
SciPy Version:        1.16.0
Matplotlib Version:   3.10.0
ARC Version:          3.9.0
Python Version:       3.11.8
Python Install Path:  ~\Miniconda3\envs\rq
Platform Info:        Windows (AMD64)
CPU Count and Freq:   12 @ 3.60 GHz
Total System Memory:  128 GB
```

CHAPTER

SIXTEEN

MODELING SATURATED ABSORPTION SPECTROSCOPY

In this notebook we use the steady-state solver in rydiqule to model saturated absorption spectroscopy (SAS). In SAS, a weak probing field and a strong counter-propagating pumping field probe transitions withing a Doppler-broadened atomic ensemble. The probe field absorption is monitored as a function of frequency, while the pumping field produces sub-Doppler features due to the Lamb dip effect.

Modeling this experiment in rydiqule is not a trivial task and involves some key assumptions. The difficulty lies in the fact that rydiqule does not allow multiple distinct fields in a single coupling. In other words, every coupling in rydiqule must be expressed as a single field. Normally, if you want to have two fields in a single coupling, you must define a single time-dependent field that then uses the time-solver.

In SAS, the probe and coupling fields only differ in their propagation, and therefore the sign of the Doppler shift the moving atoms see. A full model of the system would require defining a time-dependent coupling where the frequencies of the two fields would depend on the velocity class being solved. This is difficult to do in rydiqule and would be much slower than a steady-state model.

Here we make an approximate model of SAS that assumes a weak probe that does not influence the optical pumping of the system. Under this assumption, we can solve the system assuming only the pump field is present, then look at the relevant ground state population as a proxy for the observed probe absorption.

This notebook can be downloaded here.

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

```
%load_ext line_profiler
```

```
import numpy as np
import rydiqule as rq
import matplotlib.pyplot as plt
from arc import Wigner6j
```

Start my defining the states of our manifolds. We use the string labeling of states so that we can use regex matching patterns when defining couplings later on.

Note that we do not consider the magnetic-sublevel structure here beyond accounting for appropriate level degeneracies.

```
g_list = [1,2]
e_list = [0,1,2,3]
g = ('g', g_list)
e = ('e', e_list)
```

This defines width of the Doppler distribution.

```
k = 1/780.24e-3
kb = 1.38e-23
mass = 1.41e-25
```

(continues on next page)

(continued from previous page)

```
temp = 300
vP = np.sqrt(2*kb*temp/mass) # most probable speed, 3D
print(f'Most probable speed: {vP:.2f} m/s')
```

```
Most probable speed: 242.33 m/s
```

```
s = rq.Sensor([g,e], vP=vP)
print(s.states)
print(len(s.states))
```

```
[('g', 1), ('g', 2), ('e', 0), ('e', 1), ('e', 2), ('e', 3)]
6
```

We need to define the relative strength of the different hyperfine states. We use ARC's `Wigner6J` with Eq. 41 of Steck's Rubidium 87 D line Data. Note that these prefactors normalize as $\sum_{F'} S_{FF'} = 1$.

```
def Sff(J, F, Jp, Fp, I=3/2):

    return (2*Fp+1)*(2*J+1)*Wigner6j(J, Jp, 1, Fp, F, I)**2
```

Here we calculate the effective Clebsch-Gordon coefficients for the Pump couplings between the ground and excited states.

```
cg = {(('g',Fg), ('e', Fe)): Sff(1/2, Fg, 3/2, Fe)
      for Fg in g_list
      for Fe in e_list
      if Sff(1/2, Fg, 3/2, Fe) != 0.0
      }
print(f'There are {len(cg):d} total couplings')
```

```
There are 6 total couplings
```

We define a coupling group that maps our single coupling onto all the relevant couplings between the manifolds.

```
dets = np.linspace(-310, 250, 561)
rabi_freq = 6

k = 1/780.241e-3 #wavelength in um to get units right
kp = 2*np.pi*k*np.array([1,0,0])

s.add_coupling((g, e), rabi_frequency=2*np.pi*rabi_freq, detuning=2*np.pi*dets,␣
↪coupling_coefficients=cg, kvec=kp, label='pump')
```

Here we define the "Clebsch-Gordon"-like coefficients, but for the dephasings between states.

```
g_manifold_degen = np.sum([2*F+1 for F in g_list])
cgg = {(('e',Fe), ('g', Fg)) : (2*Fg+1)/g_manifold_degen
       for Fe in e_list
       for Fg in g_list
      }
print(f'There are {len(cgg):d} total couplings')
```

```
There are 8 total couplings
```

```
gamma = 2*np.pi*6.0666
s.add_decoherence((e, g), gamma, coupling_coefficients=cgg, label='HFS_gamma')
```

Next, we define the hyperfine level shifts. The excited states are all defined relative to the center of mass frequency (ie the fine structure frequency). The $F_g = 1$ ground state is defined relative to the $F_g = 2$ ground state.

```
hfs_shifts = {('e', 3): 2*np.pi*193.7407,
              ('e', 2): -2*np.pi*72.9112,
              ('e', 1): -2*np.pi*229.8518,
              ('e', 0): -2*np.pi*302.0738,
              ('g', 1): -2*np.pi*6834.682610904290}
```

```
s.add_energy_shifts(hfs_shifts)
```

Since this measurement is typically done in a vapor cell, we add transit broadening. In the doppler-free solve shown below, this prevents all population from being pumped into the stretch state.

```
s.add_transit_broadening(2*np.pi*0.1, repop={('g',Fg): (2*Fg+1)/g_manifold_degen␣
↪for Fg in g_list})
```

```
%%time
solRb87D2 = rq.solve_steady_state(s)
```

```
CPU times: total: 375 ms
Wall time: 134 ms
```

```
pops = rq.get_rho_populations(solRb87D2)
```

```
pops.shape
```

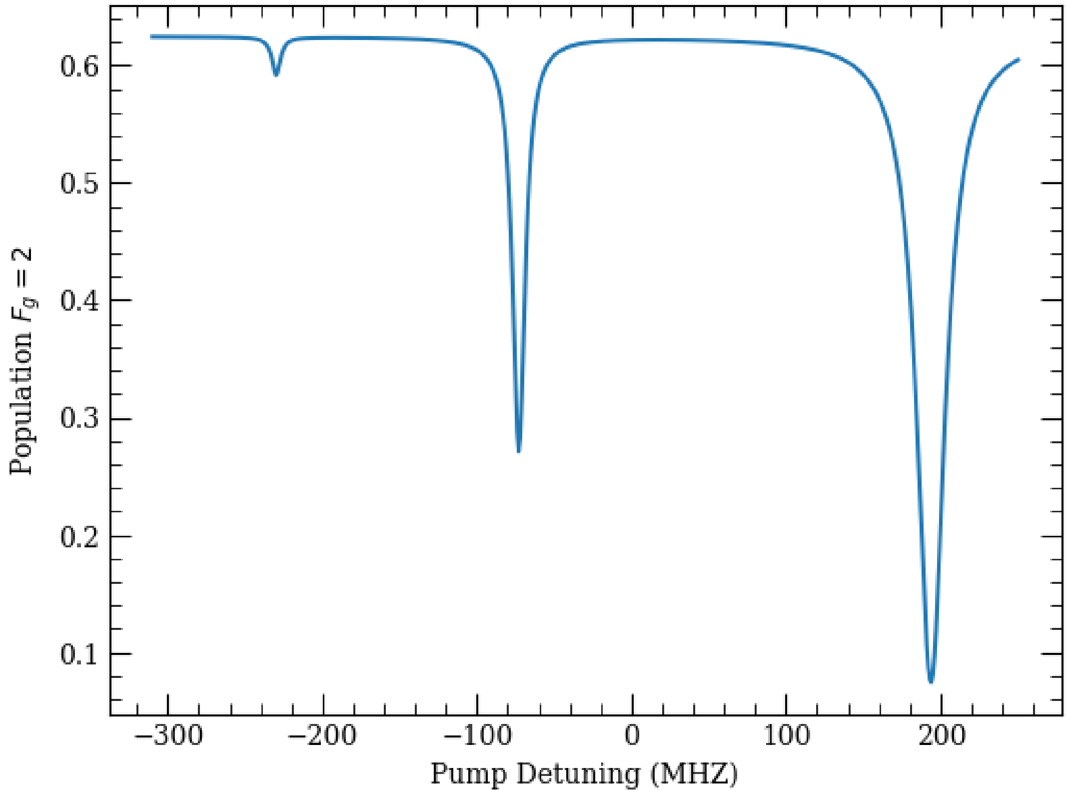
```
(561, 6)
```

```
s.axis_labels()
```

```
['pump_detuning']
```

```
fig, ax = plt.subplots(1)

ax.plot(dets, pops[:,1])
ax.set_xlabel('Pump Detuning (MHZ)')
ax.set_ylabel('Population $F_g=2$')
```

```
Text(0, 0.5, 'Population $F_g=2$')
```



We now solve the system with doppler shifts. In order to correctly simulate SAS, we will not allow rydiqule to average the velocity classes together. We have to do that ourselves. We also provide a fixed, fine grid of velocity classes to solve for as a simplification of the analysis later on.

```
dops = np.linspace(-2.5,2.5,1201)
dop_meth = {'method':'direct', 'doppler_velocities':dops}
```

```
%%time
solRb87D2dop = rq.solve_steady_state(s, doppler=True, doppler_mesh_method=dop_meth,
↪ sum_doppler=False, weight_doppler=True)
```

```
CPU times: total: 11 s
Wall time: 9.88 s
```

```
dop_popFg2 = rq.get_rho_populations(solRb87D2dop)[:,:,1]
```

```
dop_popFg2.shape
```

```
(1201, 561)
```

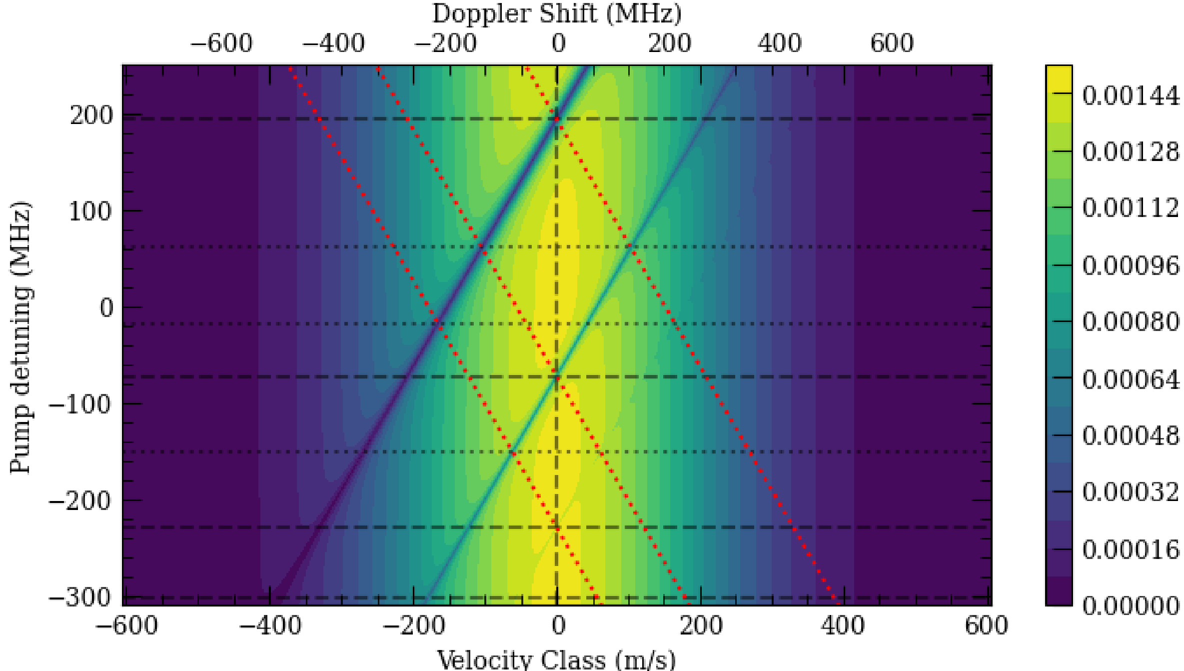
Here we define the expected detunings of the observable hyperfine transitions and the cross-over peaks.

```
hfs_resonances = np.fromiter(hfs_shifts.values(), dtype=float)[:-1]
print(hfs_resonances)
co_pairs = [(0,1),(1,2),(0,2)]
co_resonances = np.array([(hfs_resonances[i]+hfs_resonances[j])/2 for (i,j) in co_
↪pairs])
print(co_resonances)
```

```
[ 1217.30871964  -458.11458057 -1444.20145259 -1897.98566184]
[ 379.59706954 -951.15801658 -113.44636647]
```

```
fig, ax = plt.subplots(1, figsize=(8,4))

D, V = np.meshgrid(dets, dops*vP)

CS = ax.contourf(V, D, dop_popFg2, levels=20)
CB = fig.colorbar(CS, ax=ax, orientation='vertical')

def VtoMHz(v):
    return v*k

def MHztoV(MHz):
    return MHz/k

secax = ax.secondary_xaxis('top', functions=(VtoMHz, MHztoV))
secax.set_xlabel('Doppler Shift (MHz)')

for res in hfs_resonances:
    ax.axhline(res/2/np.pi, c='k', ls='--', alpha=0.5)

for co in co_resonances:
    ax.axhline(co/2/np.pi, c='k', ls=':', alpha=0.5)

ax.axvline(0, c='k', ls='--', alpha=0.5)

probe_res = -k*dops*vP
for res in hfs_resonances[:-1]:
    ax.plot(dops*vP, probe_res + res/2/np.pi, 'r:')

ax.set_xlabel('Velocity Class (m/s)')
ax.set_ylabel('Pump detuning (MHz)')
ax.set_ylim((dets.min(), dets.max()))
```

```
(-310.0, 250.0)
```



In the above plot, we show the population of the $F_g = 2$ ground state versus velocity class and pump detuning. The depleted $F_g = 2$ populations corresponding to resonances with the $F_e = (1, 2, 3)$ excited states can be seen as lines with positive slope. The horizontal gray dashed lines are the 0 velocity class resonances for the $F_g = 2, F_e = (1, 2, 3)$ transitions. The horizontal gray dotted lines are the cross-over peak locations.

The red dotted lines show the same transition resonances visible in the population depletion, but for the counter-propagating probe (which experiences the opposite doppler shift).

Our counter-propagating probe, if sufficiently weak, does not influence the $F_g = 2$ ground state population itself. Therfore, its absorption is proportional to the $F_g = 2$ population in the velocity classes that have resonant couplings to excited states. In other words, it can only probe ground state population using a dipole-allowed transition, and which velocity classes satisfy that condition depends on overall detuning of the probe/pump.

Therefore, we need to sample our doppler solves above for the resonant velocity classes as a function of detuning (ie the red dotted lines).

$$\delta = f_i - kv$$

We also want to account for the fact the probing transition is finite width, and weight nearby velocity classes by the lorentzian response.

```
dop_shift_step = np.diff(dops*vP*k)[0]
print(f'Velocity class step size {dop_shift_step:.3f} MHz')
```

```
Velocity class step size 1.294 MHz
```

```
def lorentzian(f, f0, Γ):
    return 1/np.pi*(Γ/2)/((f-f0)**2 + (Γ/2)**2)
```

```
lor_window = np.linspace(-10, 10, 25)
print(lor_window)
```

```
[-10.          -9.16666667  -8.33333333  -7.5         -6.66666667
  -5.83333333  -5.          -4.16666667  -3.33333333  -2.5
  -1.66666667  -0.83333333   0.           0.83333333   1.66666667
```

(continues on next page)

(continued from previous page)

```
 2.5          3.33333333   4.16666667   5.           5.83333333
 6.66666667   7.5          8.33333333   9.16666667  10.        ]
```

We calculate the weighting factors, and confirm they sum to approximately 1. Note that the doppler sampling is fairly course relative to the transition's linewidth, so our sampling will always be rough.

```
lor_weights = lorentzian(lor_window, 0, gamma/2/np.pi)
print(np.sum(lor_weights))
```

```
0.9836287684107903
```

```
resonances = hfs_resonances[:-1]/2/np.pi
print(resonances)
probe_classes_norm = (resonances[:,None,None] - dets[None,None,:] + lor_
↪window[None,:,None])/k/vP
print(probe_classes_norm.shape)
```

```
[ 193.7407  -72.9112 -229.8518]
(3, 25, 561)
```

Need to find the closest solved velocity class for each resonant point.

```
def find_closest_ind(x):

    min_ind = np.argmin(np.abs(dops-x))

    return min_ind

np_find_closest_ind = np.vectorize(find_closest_ind)
```

```
res_inds = np_find_closest_ind(probe_classes_norm)
print(res_inds.shape)
```

```
(3, 25, 561)
```

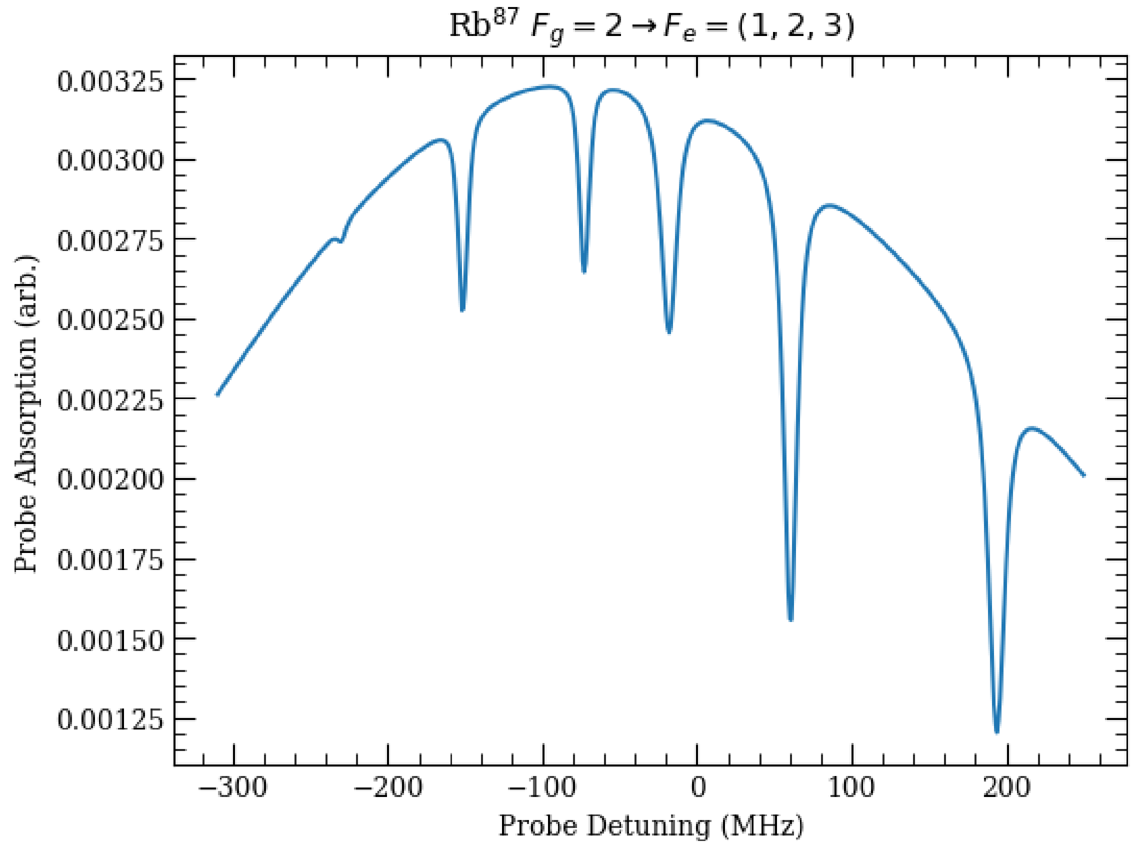
Now we slice out the relevant velocity classes for each detuning, weight them, and sum.

```
SAS_pops = dop_popFg2[res_inds,np.arange(len(dets))]  # selects indeces along␣
↪doppler axis for each detuning
print(SAS_pops.shape)
SAS_weighted_pops = SAS_pops*lor_weights[None,:,None] # weight the classes near␣
↪each resonance by the lorentzian response of the probe
SAS_probe = np.sum(SAS_weighted_pops, axis=(0,1))
print(SAS_probe.shape)
```

```
(3, 25, 561)
(561,)
```

```
fig, ax = plt.subplots(1)

ax.plot(dets, SAS_probe)
ax.set_xlabel('Probe Detuning (MHz)')
ax.set_ylabel('Probe Absorption (arb.)')
ax.set_title('Rb$^{87}$ $F_g=2\\rightarrow F_e=(1,2,3)$')
```

```
Text(0.5, 1.0, 'Rb$^{87}$ $F_g=2\\rightarrow F_e=(1,2,3)$')
```




```
rq.about()
```

```
        Rydiqule
    ================

Rydiqule Version:     2.1.0.dev11+gf4e837b.d20250222
Installation Path:    ~\src\Rydiqule\src\rydiqule

      Dependencies
    ================

NumPy Version:        1.26.4
SciPy Version:        1.12.0
Matplotlib Version:   3.8.0
ARC Version:          3.6.0
Python Version:       3.11.8
Python Install Path:  ~\Miniconda3\envs\rq
Platform Info:        Windows (AMD64)
CPU Count and Freq:   12 @ 3.60 GHz
Total System Memory:  128 GB
```

PYTHON MODULE INDEX

r

rydiqule, 111
rydiqule.arc_utils, 111
rydiqule.atom_utils, 122
rydiqule.cell, 135
rydiqule.doppler_exact, 175
rydiqule.doppler_utils, 177
rydiqule.exceptions, 186
rydiqule.experiments, 188
rydiqule.rydiqule_utils, 190
rydiqule.sensor, 191
rydiqule.sensor_solution, 228
rydiqule.sensor_utils, 237
rydiqule.slicing, 246
rydiqule.slicing.slicing, 246
rydiqule.solvers, 250
rydiqule.stack_solvers, 253
rydiqule.stack_solvers.cyrk_solver, 253
rydiqule.stack_solvers.scipy_solver, 254
rydiqule.timesolvers, 256